НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

Факультет інформатики та обчислювальної техніки
(повна назва інституту/факультету)

Кафедра автоматизованих систем обробки інформації та управління
(повна назва кафедри)

«На правах рукопису»

УДК *004.434*

«До захисту допущено»
В.о. завідувача кафедри
_____ *О. А. Павлов*
(підпис) (ініціали, прізвище)
«_____» _____ 2020 р.

Магістерська дисертація

на здобуття ступеня магістра

за освітньо-науковою програмою *«Інженерія програмного забезпечення комп'ютеризованих систем»*

зі спеціальності *121 «Інженерія програмного забезпечення»*

на тему *«Методи та засоби для автоматизованого тестування програмного забезпечення»*

Виконав *студент VI курсу ФІОТ, групи ІП-81мн*
Духін Владислав Олексійович _____
(прізвище, ім'я, по-батькові) (підпис)

Керівник *доц., к.т.н., викладач кафедри АСОІУ, І. П. Муха* _____
(посада, науковий ступінь, вчене звання, ініціали, прізвище) (підпис)

Рецензент _____ _____
(посада, науковий ступінь, вчене звання, ініціали, прізвище) (підпис)

Засвідчую підписом, що у цій магістерській дисертації немає запозичень з праць інших авторів без відповідних посилань.

Студент _____
(підпис)

Київ – 2020 року

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»
Факультет інформатики та обчислювальної техніки
Кафедра автоматизованих систем обробки інформації та управління

Рівень вищої освіти *другий (магістерський)*
(повна назва)

Спеціальність *121 «Інженерія програмного забезпечення»*
(код та назва)

Освітньо-наукова програма *«Інженерія програмного забезпечення комп'ютеризованих систем»*
(повна назва)

ЗАТВЕРДЖУЮ
В.о. завідувача кафедри
_____ *О. А. Павлов*
(підпис) (ініціали, прізвище)
«_____» _____ 2020 р.

ЗАВДАННЯ
на магістерську дисертацію студенту

Духіну Владиславу Олексійовичу
(прізвище, ім'я, по-батькові)

1. Тема дисертації *Методи та засоби для автоматизованого тестування програмного забезпечення*
науковий керівник *к.т.н., доц., викладач кафедри АСОІУ, Муха Ірина Павлівна*
(посада, науковий ступінь, вчене звання, ПІБ)
затверджені наказом по університету *№ 910-с* від «*24*» *березня* 2020 р.

2. Термін подання студентом дисертації «*27*» *квітня* 2020 р.

3. Об'єкт дослідження *Процес автоматизованого тестування програмного забезпечення*

4. Предмет дослідження *Методи та засоби створення сценаріїв автоматизованого тестування*

5. Перелік завдань, які необхідно розробити *Дослідити наявні методи та засоби побудови сценаріїв автоматизованого тестування; дослідити наявні засоби автоматизації тестування; удосконалити методи та засоби створення сценаріїв автоматизованого тестування шляхом застосування підходів метапрограмування; виконати експериментальні дослідження характеристик запропонованих рішень.*

6. Орієнтовний перелік графічного матеріалу *схема-структурна послідовності побудови сценарію автоматизованого тестування, схема-структурна архітектури програмного забезпечення*

7. Орієнтовний перелік наукових публікацій *«Предметно-орієнтована мова для опису сценаріїв автоматизованого тестування», «Застосування предметно-орієнтованої мови для розв'язання проблем автоматизації модульного тестування»*

8. Консультанти розділів дисертації

Розділ	Прізвище, ініціали та посада консультанта	Підпис та дата	
		завдання видав	завдання прийняв

9. Дата видачі завдання «____» _____ 2020 р.

Календарний план

№ з/п	Назва етапу виконання магістерської дисертації	Термін виконання	Примітка
1	*Ознайомлення з предметною областю, вивчення літератури*	*30.11.2018*	
2	*Аналіз наявних методів та засобів автоматизованого тестування*	*28.02.2019*	
3	*Постановка та формалізація задачі дослідження*	*15.03.2019*	
4	*Аналіз вимог до програмного забезпечення*	*29.03.2019*	
5	*Розробка методів та засобів для розв'язання поставлених задач*	*31.05.2019*	
6	*Розробка архітектури програмного забезпечення*	*25.10.2019*	
7	*Виконання експериментальних досліджень*	*20.12.2019*	
8	*Оформлення пояснювальної записки*	*20.04.2020*	
9	*Подання дисертації на попередній захист*	*26.04.2020*	
10	*Подання дисертації на рецензію*	*11.05.2020*	
11	*Подання дисертації на захист*	*19.05.2020*	

Студент _____ *Духін В. О.*
(підпис) (прізвище та ініціали)

Науковий керівник _____ *Муха І. П.*
(підпис) (прізвище та ініціали)

РЕФЕРАТ

Магістерська дисертація: 91 с., 19 рис., 6 табл., 3 додатки, 28 джерел

Актуальність теми. Одним з найважливіших етапів життєвого циклу розробки програмного забезпечення є перевірка працездатності написаного програмного коду та відповідності поставленим вимогам. Ручне тестування програмного забезпечення займає багато часу, тому дослідження методів та засобів для автоматизації тестування є актуальною задачею. Важливою складовою процесу автоматизації є складання відповідного сценарію тестування, який охоплював би усі можливі варіанти використання програмного забезпечення, що тестується. Проте, внаслідок складності написання сценаріїв тестування мовами програмування та за відсутності надійних засобів для автоматизованого тестування у програмній платформу Node.js, ускладняється процес тестування програмного забезпечення. Тому актуальною є задача розробки мови опису сценаріїв автоматизованого тестування та фреймворку тестування для програмної платформи Node.js.

Мета досліджень. Підвищити ефективність процесу автоматизації тестування та спростити написання сценаріїв тестування шляхом розробки предметно-орієнтованої мови програмування, що передбачає використання металінгвістичних абстракцій та словникового запасу базової мови програмування.

Для реалізації поставленої мети були сформовані **наступні завдання**:

– дослідити наявні методи та засоби побудови сценаріїв автоматизованого тестування;

– дослідити наявні засоби автоматизації тестування для програмної платформи Node.js;

– удосконалити методи та засоби створення сценаріїв автоматизованого тестування шляхом застосування підходів метапрограмування;

– розробити фреймворк автоматизованого тестування для програмної платформи Node.js;

– виконати експериментальні дослідження характеристик запропонованих рішень.

Об’єкт досліджень. Процес автоматизованого тестування програмного забезпечення.

Предмет досліджень. Методи та засоби створення сценаріїв автоматизованого тестування.

Методи досліджень. Емпіричні дослідження, системний аналіз, абстрагування, структурно-генетичний аналіз, розробка програмного забезпечення.

Наукова новизна отриманих результатів полягає в удосконаленні методу створення сценаріїв автоматизованого тестування шляхом структуризації такого сценарію та представлення окремих полів сценарію й тестових випадків у форматі модифікованих S-виразів, які є металінгвістичними абстракціями, заданими з використанням словникового запасу базової мови програмування. Для формування такого сценарію була розроблена спеціальна предметно-орієнтована мова програмування, яка схожа на звичайну розмовну мову, а отже є простим та зрозумілим інструментом створення сценаріїв автоматизованого тестування.

Практичне значення отриманих результатів полягає у застосуванні розробленого фреймворку автоматизованого тестування у проєктах, що потребують автоматизації модульного тестування.

Апробація. Результати роботи доповідались на «IV всеукраїнській наково-практичній конференції молодих вчених та студентів «Інформаційні системи та технологій управління» (ІСТУ-2020).

Публікації. Наукові положення опубліковані в тезах наукової конференції «IV всеукраїнська науково-практична конференція молодих вчених та студентів «Інформаційні системи та технологій управління» (ІСТУ-2020).

Ключові слова: ФОРМАЛЬНА МОВА, ПРЕДМЕТНО-ОРІЄНТОВАНА МОВА, S-ВИРАЗ, ГРАМАТИКА, СИНТАКСИС, ФРЕЙМВОРК, РБНФ

ABSTRACT

Topic: «Automated Software Testing Methodologies»

Master's degree thesis: 91 pages, 19 figures, 6 tables, 3 attachments, 28 sources.

Relevance of the topic. One of the most important stages of the software development life cycle is the check of the program's code operational performance and check of its conformity to the requirements set. Manual testing of software takes a lot of time, so the study of automated software testing methodologies is a high priority task. An important component of the automation process is the creation of an appropriate test scenario that would cover all possible use cases of the software being tested. But as a result of the complexity of writing test scripts in programming languages and the lack of reliable tools to automate testing in Node.js, the software testing process becomes more complicated. That is why it is important to develop a language for describing automated testing scenarios and a testing framework for Node.js.

Research objective. Increase the efficiency of the test automation process and simplify the creation of test scenarios by developing a domain-specific programming language that uses metalinguistic abstractions and vocabulary of the basic programming language.

The following objectives have been formulated to achieve this research objective:

– explore existing methods and ways of creating automated test scenarios;

– explore existing ways to automate testing in Node.js;

– improve methods and ways of creating automated test scenarios through the use of metaprogramming;

– develop a test automation framework for Node.js;

– perform experimental research on the characteristics of the proposed solutions.

The object of research. The process of automated software testing.

The subject of research. Methods and techniques for creating automated test scenarios.

The scientific innovation of the obtained results lies in the improvement of the automated test scenario creation method by means of scenario structuring and presentation of

separate scenario fields in the format of modified S-expressions, which are metalinguistic abstractions. To create such test scenarios, a special domain-specific programming language has been developed that is similar to an ordinary human language, which means that test scenarios in this language are simple and clear to understand.

The practical value of the obtained results lies in the use of the developed test automation framework in projects that require the automation of unit tests.

Approbation. The results of the master's degree thesis were reported at the student conference, which was held in Kiev, Ukraine in 2020.

Scientific publications. The scientific papers were published in a collection of materials from the student conference held in Kiev, Ukraine in 2020.

Keywords: FORMAL GRAMMAR, DOMAIN-SPECIFIC LANGUAGE, S-EXPRESSION, SYNTAX, TEST AUTOMATION FRAMEWORK, UNIT TESTING, EBNF

ЗМІСТ

ВСТУП

На початку розвитку індустрії розробки програмного забезпечення, його тестуванню приділялась опосередкована увага. Інженери з тестування застосовували ручне тестування програмного забезпечення з метою визначення його працездатності та наявності різноманітних дефектів бізнес-логіки або інтерфейсу користувача. Проте в процесі інтенсивного розвитку технологій, мов програмування та підвищення складності програмних систем поставали нові проблеми, що вимагали вдосконалення методів та засобів проведення тестування. Ручне тестування складних систем займало багато часу та потребувало значних витрат зусиль на його проведення, а сценарії тестування багаторазово виконувались і найчастіше не потребували значних змін. Актуальним постало питання автоматизації процесів тестування і контролю якості програмного забезпечення.

Сьогодні однією з основних сфер розвитку тестування програмного забезпечення є його автоматизація. Згідно зі звітом «Pulse of the Profession 2019» [1], більшість команд працює за методологією гнучкої або гібридної розробки програмного забезпечення (такими як SCRUM та Kanban). Відповідно, в ітеративному процесі гнучкої розробки, контролю якості розробки надається високий пріоритет для забезпечення швидкої та ефективної роботи. На ряду із ручним тестуванням, команди все частіше використовують автоматизоване тестування, що дозволяє їм спростити виконання сценаріїв тестування і переважно зменшити необхідність в ручному тестуванні. Автоматизація тестів дозволяє їм дізнаватись про дефекти в програмах за лічені хвилини після внесення змін у вихідний код, тоді як ручне тестування зайняло б години та дні. Тенденції світу показують, що темпи розробки програмного забезпечення з кожним роком зростають, що робить автоматизацію тестування важливим питанням в таких сферах як неперервна інтеграція (англ. continuous integration) та доставлення (англ. continuous delivery) програмного забезпечення.

Автоматизоване тестування являється ключовим підходом у таких методологіях розробки як екстремальне програмування та розробка керована тестуванням. В цих методологіях, сценарії тестування створюються ще до початку розробки певного модуля програмного продукту, а з початком розробки, сценарії розширюються по мірі розвитку продукту. Такий підхід дозволяє командам акцентувати увагу на архітектурі системи, взаємодії її компонент та аналізі проблемної галузі, а великий відсоток покриття коду тестами дозволяє виявляти дефекти одразу після внесення змін у вихідний код.

Вперше термін «автоматизоване тестування» було запропоновано Фредеріком Бруксом [2] у його книзі «Міфічний людино-місяць, або як створюють програмні системи». В ній він навів вичерпні приклади потреби тестування для забезпечення концептуальної цілісності програмного продукту у розрізі застосування модульного тестування.

Міжнародна рада контролю якості програмного забезпечення (англ. International Software Testing Qualifications Board) визначає автоматизоване тестування як використання програм для підтримки процесів тестування, таких як менеджмент сценаріїв тестування, розробка дизайну сценаріїв та перевірка результатів тестування [3]. Автоматизація тестування дозволяє командам швидко та багаторазово виконувати тести, які у випадку ручного тестування були б дуже трудомісткими до виконання. Автоматизація тестування знаходить широке застосування для проведення регресивного тестування – тестування програмного забезпечення після модифікації для визначення виникнення дефектів або змін у немодифікованих частинах програмного забезпечення як наслідок внесених змін [3].

Метою автоматизованого тестування є виконання сценаріїв тестування, що були підготовлені заздалегідь інженерами з тестування за допомогою спеціальних методологій, перевірка і порівняння результатів виконання з еталонами або очікуваними результатами. Такі перевірки виконуються із застосуванням спеціалізованих інструментів та фреймворків, окремих від розробленого програмного забезпечення.

Фреймворки надають комплексне рішення для проведення тестування і служать платформою для інженера з тестування. Вони надають програмну інфраструктуру, середовище тестування та певне архітектурне рішення для опису сценаріїв, їх організації, виконання та валідації. Своєю чергою спеціалізовані інструменти для тестування можуть бути орієнтовані лише на певну систему, середовище тестування або процес. Прикладом такого інструменту може бути статичний аналізатор коду, що дозволяє ідентифікувати помилки синтаксичного та типографічного характеру.

Багато організацій розглядають автоматизацію тестування програмного забезпечення як рішення для зменшення витрат на тестування та пришвидшення циклу розробки. Однак рішення про необхідність автоматизованого тестування може бути невдалим, якщо автоматизація тестування не буде застосована в правильному контексті та відповідному до потреб організації підході. Так, наприклад, повторне виконання сценаріїв тестування та повторне застосування однакових методів тестування може призвести до того, що деякі дефекти програмного забезпечення залишаться незнайденими. Цей парадокс був названий як «парадокс пестицидів» і вперше описаний Боризом Бейзером у книжці «Software Testing Techniques» [4], де він порівняв цю стратегію тестування з повторною обробкою сільськогосподарських полів речовинами проти шкідників.

Іншою проблемою автоматизації тестування є необхідність навичок розробки програмного забезпечення в інженерів з тестування, оскільки сценарії тестування найчастіше описуються у вигляді програмного коду, який згодом автоматизовано виконується у середовищі тестування. Погіршує ситуацію складність сучасних систем, які мають розгалужену архітектуру, програмні залежності між компонентами системи і можуть потребувати створення та налаштування спеціальної інфраструктури для своєї роботи. Необхідність навичок програмування в інженерів з тестування може бути нівельована покладанням написання сценаріїв тестування на розробників програмного

забезпечення, але в більшості випадків це призведе до додаткових витрат або сповільнення темпів розробки.

Дослідження 78 різноманітних джерел показали, що проблеми невдалого застосування автоматизованого тестування можуть бути поділені на п’ять груп факторів: пов’язані з програмним забезпеченням та його архітектурою, пов’язані з підходами до тестування та написання сценаріїв тестування, пов’язані з проблемами інструментів для проведення тестування, людські й організаційні фактори та сукупність факторів різних груп [5].

Змінити ситуацію та вирішити сучасні проблеми тестування може дослідження наявних методів та засобів для автоматизованого тестування, вдосконалення цих методів та засобів або створення нових підходів до автоматизації тестування. Так, наприклад, створення мови опису сценаріїв тестування, яка була б одночасно зрозумілою та простою до написання, але в той же час являлась мовою програмування для виконання цих сценаріїв, допомогло б у вирішенні проблеми необхідності навичок програмування в інженерів з тестування, а отже більша кількість людей могла б бути залучена до процесу тестування. Вдосконалення методологій та підходів до тестування могло б значним чином вплинути на структуру сценаріїв тестування та знизити потребу у їх постійному редагуванні.

Метою даної магістерської роботи є підвищення ефективності процесу автоматизації тестування та спрощення процесу створення сценаріїв автоматизованого тестування шляхом використання металінгвістичних абстракцій та словникового запасу базової мови програмування.

Актуальністю цих досліджень, а також наукової діяльності у наведеній предметній галузі є те, що попри інтенсивний розвиток програмного забезпечення та збільшення його складності, засоби автоматизованого тестування рідко адаптуються до нових потреб бізнесу, змін у мовах програмування, та не надають гнучкості у виборі підходів до проведення тестування. Фреймворки тестування програмної платформи

Node.js здебільшого фокусуються на певних методологіях тестування, а поєднання декількох типів тестування у одному проєкті потребує встановлення додаткових програмних залежностей та їх інтеграції один з одним.

Об’єктом досліджень даної магістерської роботи є процес автоматизованого тестування програмного забезпечення. В роботі акцент надається дослідженню методів та засобів створення сценаріїв автоматизованого тестування, виразності та лаконічності мов, що описують їх та швидкодії засобів для проведення автоматизованого тестування, що є предметом цих досліджень.

Для досягнення мети магістерської дисертації необхідно виконати наступні завдання:

– дослідити наявні методи та засоби побудови сценаріїв автоматизованого тестування;

– дослідити наявні засоби автоматизації тестування для програмної платформи Node.js;

– удосконалити методи та засоби створення сценаріїв автоматизованого тестування шляхом застосування підходів метапрограмування;

– розробити фреймворк автоматизованого тестування для програмної платформи Node.js;

– виконати експериментальні дослідження характеристик запропонованих рішень.

1 ОГЛЯД МЕТОДІВ ТА ЗАСОБІВ АВТОМАТИЗАЦІЇ ТЕСТУВАННЯ

1.1 Класична модель тестування

Ще до появи гнучких методологій розробки програмного забезпечення, таких як SCRUM, автоматизоване тестування широко використовувалось для спрощення та пришвидшення процесів тестування. Проте автоматизовані тести були занадто трудомісткими у розробці та вимагали детального розуміння архітектури створеної системи для їх написання. Однією з причин цьому було те, що команди розробників та інженерів з тестування намагались автоматизувати сценарії тестування одночасно на різних рівнях деталізації системи, що призводило до низького покриття коду тестами та зниженню їх ефективності.

За класифікацією, що була вперше запропонована Майком Коном у його роботі «Succeeding with Agile» [6], стратегія автоматизації тестування може бути поділена на три рівні:

- модульне тестування;
- сервісне тестування;
- тестування користувацького інтерфейсу.

Ці рівні утворюють так звану «класичну піраміду тестування», що зображена на рисунку 1.1, де кожні наступні рівні тестів, починаючи з модульних, розширюють тестові сценарії нижчих рівнів, а також додають нові сценарії інтеграції частин системи. Загальна кількість тестів певного рівня залежить від позиції цього рівня у «піраміді». Особливістю піраміди є те, що з підвищенням рівня тестування підвищуються й витрати часу та зусиль на розробку і підтримку тестових сценаріїв. Тести вищого рівня відрізняються значною крихкістю, оскільки за навіть незначної модифікації системи можуть стати недійсними у великій кількості. Проте такі сценарії тестування

узагальнюють та завершують системність тестування, і найчастіше демонструють сценарії використання розробленої системи.

Рисунок 1.1 – «Класична піраміда тестування» за Майком Коном

1.1.1 Модульне тестування

Модульне тестування є основою «піраміди тестування» та надає гарантію, що найменші частини програмного забезпечення такі як функції, методи та класи, працюють згідно з поставленими до них вимог. На думку Майка Кона [6], такі тести повинні складати основну частину автоматизованого тестування, а кількість розроблених сценаріїв тестування цього рівня має бути найбільшою. Найчастіше модульні тести створюють програмісти без участі інженерів з тестування, а сценарії цих тестів являють собою програмний код, що тестує деякий функціонал в ізоляції від інших компонентів системи. Модульні тести є деталізованими та із використанням методу регресивного тестування дозволяють ефективно виявляти дефекти вихідного коду. Внаслідок ізольованості тестування вдається однозначно визначити місце, де з’явився знайдений дефект.

Ізольованість компонентів системи під час модульного тестування здебільшого виконується методом заміни залежних компонентів, тобто за допомогою реалізації спеціального програмного компонента, який заміщує собою при тестуванні ті компоненти, які викликає вихідний компонент або від яких тим чи іншим чином

залежить [3]. Ізолювання компонентів гарантує уникнення побічних ефектів під час тестування та звужує зону пошуку дефекту при його виявленні, адже фактично окремий компонент під час тестування не залежить від інших компонентів. В деяких випадках програмісти можуть нехтувати ізолюванням компонентів системи при модульному тестуванні, через здебільшого складність реалізації таких сценаріїв тестування, відсутність надійних засобів для підробки або заміни залежних компонентів в інструментах та фреймворках автоматизованого тестування, або ж у випадках коли побічними ефектами можливо усвідомлено знехтувати (наприклад, у випадку використання вбудованих компонентів мови програмування).

За принципом ізоляції компонентів Джей Філдс у своїй книжці «Working Effectively with Unit Tests» [7] поділив модульне тестування на два типи:

– самітницький модульний тест (англ. solitary unit test) – тест, об’єкт тестування якого повністю ізольований від інших компонентів системи;

– товариський модульний тест (англ. sociable unit test) – тест, об’єкт тестування якого може взаємодіяти зі справжніми компонентами системи: частково або повністю.

Існує також поширена проблема ізоляції компонентів в мовах програмування, що реалізують можливість обмеження прямого доступу до компонентів системи з метою приховання їх конкретної реалізації [8, 9]. В таких випадках якщо компонент залежить від компонента, що обмежений у доступі, то можливо знехтувати цією залежністю, або змінити режим доступу до компонента за допомогою інструментів мови програмування для його подальшої ізоляції. Останнє є вкрай небажаним, адже порушує принципи цієї мови програмування і створює ситуацію, коли середовище тестування відрізняється від виробничого середовища виконання програми. Компоненти, що обмежені у прямому доступі, слід розглядати як деталі реалізації системи, а саме тому тестуванням цих модулів необхідно знехтувати [9].

Джерард Мецарос у своїй книзі «xUnit Test Patterns: Refactoring Test Code» [10] виділив низку методів ізолювання компонентів системи при модульному тестуванні, які він об’єднав під терміном «двійники» (англ. test doubles). Всі ці методи мають на увазі імітацію поведінки справжнього компонента так, щоб інші компоненти системи не помітили підміни під час тестування. Принципи імітації, яку вони виконують, схожі між собою, проте сутність роботи цих компонентів на етапі виконання сценарію тестування суттєво відрізняється. Джерард виділив наступні методи імітування:

– порожній компонент (англ. dummy) – об’єкти, що застосовуються як заповнювачі у списках аргументів, але фактично ніколи не використовуються; на виклики до них не надають відповіді;

– фіктивний компонент (англ. fake) – частково реалізований компонент, поверхнево схожий на вихідний компонент; здебільшого компонент, що заповнений фіктивними даними;

– компонент-заглушка (англ. stub) – об’єкт, що відповідає на виклики до нього заготовленими відповідями, але не відповідає на такі виклики, які не були заздалегідь заготовлені; іноді компонент, що частково імітує поведінку справжнього компонента з допомогою спрощеного функціоналу;

– компонент-шпигун або компонент-посередник (англ. spy) – так само як і компоненти-заглушки є частково реалізованими, проте при викликах до нереалізованих методів передають відповідальність за відповідь до вихідного компоненту системи; часто використовуються як посередники між викликами до вихідного компоненту;

– імітований компонент або макет (англ. mock) – компонент, який повністю імітує поведінку вихідного компонента через реалізацію його інтерфейсу, проте не має реалізації справжньої функціональності, а лише імітує її.

Описані Джерардом методи підробки справжніх компонентів системи, окрім імітування роботи компонента, можуть також збирати статистику, яку згодом на етапі перевірки результатів виконання тесту можливо використовувати для валідації

поведінки системи. До таких статистик можуть відноситись кількість викликів до компонента, аргументи з якими були зроблені ці виклики, контекст виконання, стан системи тощо.

Ізолювання компонентів один від одного під час модульного тестування не тільки надає можливість більш точної ідентифікації дефектів при їх виникненні, а й пришвидшує виконання сценаріїв. Тестування найменших модулів системи в ізоляції від звернень до баз даних, файлової системи, програмного вводу та виводу, або виконання запитів до сторонніх систем дозволяє виконувати тисячі сценаріїв тестування за лічені хвилини [7, 9].

Цілей ізолювання можливо також досягти із застосуванням підходу впровадження залежностей (англ. dependency injection) – процес надання зовнішньої залежності програмному компоненту [11]. Впровадження залежностей виконується шляхом інверсії управління, коли фреймворк тестування надає всі необхідні залежності компонентам системи або розміщує їх у спеціальному середовищі тестування, де за допомогою пісочниць виконується ізолювання компонент.

Найважливішим аспектом модульних тестів є те, що вони повинні забезпечити тестування всіх нетривіальних сценаріїв використання компонентів системи, включаючи, але не обмежуючись:

– еквівалентне розмежування (англ. equivalence partitioning) – техніка тестування методом «чорного ящика» (англ. black-box testing), в якій множина значень змінної певного домену розділяється на еквівалентні за значенням підмножини, а далі сценарії тестування розробляються на основі використання одного значення з кожної підмножини [3]. Наприклад, якщо певний компонент приймає як аргумент значення у діапазоні від 1 до 10, то для складання сценарію тестування потрібно взяти одне випадкове значення з діапазону допустимих значень (наприклад, число 7) та одне з недопустимих (наприклад, число 0);

– аналіз граничних значень (англ. boundary-value analysis) – техніка тестування методом «чорного ящика», в якій сценарії тестування розробляються на основі граничних значень, тобто мінімального або максимального значення упорядкованого еквівалентного розмежування [3]. Для прикладу з першого підпункту, граничними значеннями будуть мінімальне та максимальне значення діапазону, тобто числа 1 та 10, а також мінімальні та максимальні граничні значення за межею діапазону, тобто числа 0 та 11;

– причинно-наслідковий аналіз (англ. cause-effect analysis) – техніка тестування методом «чорного ящика», в якій сценарії тестування розробляються на основі причин (входів або подразників системи) та пов’язаними з ними наслідками (виходи або реакції системи) [3]. Наприклад, сценарій тестування для розгалуженого алгоритму може містити в собі причини (певний набір вхідних даних алгоритму) за яких алгоритм обирає певну модель реакції, що буде наслідком взаємодії;

– передбачення помилки (англ. error guessing) – методика тестування за якої сценарії тестування розробляються на основі попередніх знань про знайдені у системі недоліки та/або на основі загальних знань про можливі дефекти у системах [3];

– вичерпне тестування (англ. exhaustive testing) – підхід до розробки сценаріїв тестування, при якому тест включає перебір всіх можливих комбінацій вхідних значень та передумов [3];

– нечітке тестування (англ. fuzz testing) – метод тестування програмного забезпечення, який використовується для виявлення вразливих місць безпеки компоненту системи шляхом вводу у систему або компонент великої кількості випадкових даних, які називаються нечіткими [3].

Водночас, модульні тести не повинні занадто сильно замикатись на конкретну реалізацію компонентів системи, аби бути гнучким до майбутніх змін. Сценарії тестування, що занадто близькі до виробничого коду при незначних змінах вихідного коду миттєво стають недійсними [8]. Таким чином втрачається основна перевага

модульних тестів – визначення дефектів функціональності модулів при внесенні змін до них. Для запобігання занадто великої деталізації модульних тестів, слід дотримуватись технік тестування методом «чорного ящика» та тестувати тільки ту поведінку системи, яку можливо спостерігати без знань про внутрішній устрій системи.

Одночасно з написанням модульних сценаріїв команди звертають увагу на один з основних покажчиків завершеності тестування за допомогою модульних тестів – покриття вихідного коду сценаріями тестування. Головною помилкою, яка при цьому часто допускається, є те, що команди прагнуть покрити тестами 100% вихідного коду проєкту, тестуючи не тільки бізнес-логіку, а й тривіальний код. Звичайно, що загальнодоступний інтерфейс програми повинен бути протестований, проте тестуванням тривіального, хоч і загальнодоступного, коду можливо знехтувати. Рівень тестування модулів системи повинен бути таким, щоб надавати достатній рівень впевненості у відповідності вимогам до системи.

Таким чином, в проєктах де команди застосовують підхід до написання одночасно і вихідного коду системи, і тестів до цієї системи, модульне тестування допомагає програмістам першочергово замислитись над сутністю проблеми та граничними випадками використання функціоналу, що вони розробляють. Сценарії модульного тестування не потребують багато зусиль на розробку, складають детальний опис окремих найменших компонентів системи та створюють базис для сценаріїв тестування вищого рівня.

1.1.2 Сервісне тестування

Майже кожна нетривіальна система складається з певної кількості підсистем та модулів, що взаємодіють між собою. Сьогодні типова система обробки даних може взаємодіяти з базами даних, файловою системою або робити виклики до інших систем через мережу Інтернет. Під час написання модульних тестів такі інтегровані системи, а також компоненти вихідної системи, ізолюють між собою задля пришвидшення виконання сценарію тестування. Проте у виробничому середовищі виконання, система

та всі її компоненти будуть взаємодіяти між собою, а отже ця взаємодія повинна бути протестована.

Сервісне або інтеграційне тестування (англ. integration testing) – це тестування в якому об’єктами тестування є інтерфейси системи та взаємодія між інтегрованими компонентами [3]. В залежності від архітектури системи такими співзалежними компонентами можуть бути як методи одного класу, так і низка функцій, що об’єднані спільною бізнес-логікою. Оскільки розробка різноманітних модулів системи або її підсистем виконується ізольовано та здебільшого різними командами, постає проблема подальшої інтеграції створених частин у єдину систему. Сутність інтеграційного тестування полягає в тестуванні роботи сукупності модулів як єдиної системи та перевірці, що окремо розроблені модулі працюють разом належним чином, а їх інтерфейси відповідають дизайну архітектури системи [12].

Наразі існує два методи проведення сервісного тестування:

– виклик справжніх інтегрованих компонент – перевірка бізнес-логіки системи відповідно до поставлених вимог з виконанням справжніх викликів до файлової системи, баз даних або інших компонент; сценарії при такому підході виконуються повільно оскільки потребують очікування відповіді від залежних систем, а створення сценаріїв сервісного тестування і середовища їх виконання потребує значних витрат ресурсів;

– ізоляція зовнішніх компонент – виклики до баз даних або інших зовнішніх компонент, що впливають на швидкодію тестування, повинні бути замінені на підроблені компоненти, що повністю імітують їх роботу, а виклики до внутрішніх компонентів системи повинні залишитись без змін; застосування цього методу значним чином залежить від вимог до функціонування проєкту та критичності часткової заміни зовнішніх компонент.

Переважно застосування другого методу дає більш надійні результати тестування, адже при виникненні дефекту внаслідок ізолювання сторонніх інтеграцій

вдається більш точно визначити місце виникнення проблеми [12]. Разом із застосуванням тестування контрактів взаємодії, сервісні тести, створенні методом ізоляції зовнішніх компонент, виконуються швидко та не потребують значних зусиль на їх реалізацію.

Оскільки сучасна архітектура програмних застосунків найчастіше являє собою шари абстракцій, то інтеграційне тестування існує на межі цих шарів або на межі самої системи та концептуально демонструє дії, що призводять до інтеграції програмного застосунку із зовнішніми компонентами. Найчастіше сервісні тести допомагають визначити дефекти інтерфейсів і контрактів викликів, проте можуть ідентифікувати проблеми з асинхронністю, багатонитковістю, станом перегонів, дефектами прикладного програмного інтерфейсу системи, а також головним чином визначити готовність бізнес-логіки програмного застосунку до випуску.

1.1.3 Тестування користувацького інтерфейсу

Більшість програм сьогодні мають інтерфейс користувача: вебінтерфейс, стільниковий інтерфейс, інтерфейс командного рядка або прикладний програмний інтерфейс для інтеграції зі сторонніми сервісами. Тести користувацького інтерфейсу, що знаходяться на вершині «класичної піраміди тестування», покликані симулювати дії користувача в інтерфейсі програми (натискання кнопок миші, набір тексту тощо), а інтерфейс програми повинен передбачувано реагувати на цю взаємодію. Загальна кількість сценаріїв тестування користувацького інтерфейсу серед всіх сценаріїв має бути найменшою, оскільки внесення будь-яких змін до користувацького інтерфейсу призведе до недійсності цих тестів, а створення таких сценаріїв тестування є трудомісткою задачею [6].

В залежності від мови програмування, технологій та типу інтерфейсу, тестування користувацького інтерфейсу в найліпшому випадку потребує підміни бізнес-логіки системи на «двійника» і написання модульних тестів до шару абстракції користувацького інтерфейсу. Проте не завжди під час тестування вдається розмежувати

шари абстракцій і тому допускається виконання тестування користувацького інтерфейсу у режимі наскрізного тестування – тип тестування, в якому бізнес-процеси тестуються від початку до кінця у середовищі тестування, максимально наближеному до виробничого [3]. На думку Майкла Кона, тестування користувацького інтерфейсу не відрізняється від наскрізного тестування, проте в реаліях сучасних можливостей фреймворків тестування вони є досить ортогональними поняттями.

Наскрізні тести надають найбільшої впевненості у працездатності розробленого програмного застосунку. Процес наскрізного тестування являє собою розгортання застосунку у середовищі тестування, максимально наближеному до виробничого, з усіма зовнішніми залежностями та необхідними даними. Використовуючи засоби автоматизованого тестування повністю виконуються всі сценарії використання програмного застосунку та необхідні перевірки на відповідність поставленим вимогам. Проблемою створення сценаріїв наскрізного тестування є велика трудомісткість їх створення та підтримки актуальності. Внаслідок цього, наскрізні тести можуть надавати псевдопозитивні результати або виходити з ладу з несподіваних і непередбачуваних причин. Чим складнішим є користувацький інтерфейс, тим складнішим у розробці є такий тип сценаріїв тестування. Впровадження реклами у веб-застосунки, будь-яких промо-кампаній, інших елементів або анімаційних компонент, що можуть несподівано з’явитись на вебсторінці, робить тестування інтерфейсу користувача майже неможливим.

Набагато складнішою задачею тестування користувацького інтерфейсу є перевірка, що певний графічний інтерфейс не змінився внаслідок внесених у вихідний код змін, що не повинні були змінити його. Проблема полягає в тому, що автоматизовано комп’ютер нездатний перевірити правильність зовнішнього вигляду графічного інтерфейсу, а тому ця задачі є досі невирішеною. Можливо, застосування машинного навчання або штучного інтелекту зможе певним чином розв’язати цю проблему. Наразі існують інструменти здатні робити знімки екрану користувача і порівнювати їх для

виявлення відмінностей. Але внаслідок описаних вище проблем, постає проблема коректної ідентифікації очікуваних змін і налаштування чутливості визнання інтерфейсу як такого, що був змінений випадково.

Крім того, тестування користувацького інтерфейсу системи з мікросервісною архітектурою може потребувати запуску та налаштування великої кількості компонентів системи, що може бути абсолютно неможливим в локальному середовищі тестування і потребувати спеціальних технічних застосунків на кшталт контейнеризації.

Внаслідок обмежень та проблем, що виникають при тестуванні користувацького інтерфейсу та наскрізному тестуванні, кількість сценаріїв тестування цього рівня повинна бути мінімальною. Для більшості систем достатнім буде лише тестування критичних сценаріїв використання програмного застосунку. Також можливим є проведення додаткового ручного тестування із залученням інженерів з тестування.

За таких обставин, завдяки наявності великої кількості сценаріїв тестування нижчих рівнів у «піраміді тестування», вдається гарантовано стверджувати про повну працездатність системи, а потреба у тестуванні граничних випадків використання системи не є доцільною.

1.2 Розширена модель тестування

Для більшості сучасних проєктів кількість модульних сценаріїв тестування повинна перевищувати кількість сценаріїв тестування користувацького інтерфейсу, і в цьому є сенс, адже найважливішим об’єктом тестування є бізнес-логіка програмного застосунку, а вже другочерговою задачею є тестування інтерфейсу користувача. Проте із розвитком програмного забезпечення виникають нові типи застосунків, а отже і специфічні потреби до їх тестування. Так, наприклад, тестування мобільного застосунку повинно бути більш акцентоване на інтерфейс користувача, тоді як тестування логіки взаємодії із зовнішніми компонентами є менш вагомою задачею. Для таких проєктів «піраміда тестування» може приймати різні форми: від трапеції до циліндра, в залежності від кількості сценаріїв тестування кожного рівня та конкретних потреб.

«Класична піраміда тестування» являє собою абстрактну модель автоматизованого тестування, а її структура необов’язково повинна повністю відтворюватись компаніями у своїх проєктах. Саме тому із розвитком підходів до тестування «класична піраміда» зазнала розширення і в залежності від кінцевих потреб бізнесу може містити різні рівні тестування та їх кількість. На рисунку 1.2 зображена найбільш популярна «розширена піраміда тестування». В ній додається рівень тестування контрактів взаємодії компонентів системи – тест, що перевіряє правильність та порядок повідомлень, що виникають у системі при комунікації між її компонентами.

Рисунок 1.2 – Приклад «розширеної піраміди тестування»

«Розширена піраміда тестування» та підходи до її опису в першу чергу фокусуються на цінностях для бізнесу та тих ризиках, що виникають під час життєвого циклу розробки програмного забезпечення. Вона не є і не може бути стандартизована, адже кожний проєкт вимагає унікальних підходів до проведення тестування. Замість акцентування уваги на кількості тестів, «розширена піраміда тестування» пропонує турбуватись про тип та якість тестування аби попередити можливі ризики певного бізнес-домену.

Сучасні потреби тестування в реаліях інтенсивного розвитку програмного забезпечення розкривають ряд суттєвих недоліків «класичної піраміди тестування». Але

емпіричні правила, що були впроваджені нею, залишаються актуальними й досі. «Класична піраміда» – це гарна модель з якої проєкти можуть будувати власні підходи до тестування та керуючись описаними методами створення сценаріїв тестування досягати поставленої мети тестування.

В наступних підрозділах будуть описані деякі з найпопулярніших методів тестування та рівнів «розширеної піраміди» автоматизованого тестування.

1.2.1 Тестування контрактів взаємодії

Одним з найпоширеніших сценаріїв використання «двійників» у тестуванні є підробка зовнішніх компонент системи, що розташовані на віддалених вузлах мережі Інтернет. Зазвичай, такі компоненти розробляє інша команда, їх швидкодія може суттєво відрізнятись від основної системи, а зв’язок компонент мережею Інтернет може бути ненадійним. Саме тому використання «двійників» допомагає ефективно виконувати такі сценарії сервісного тестування, проте постає проблема точності представлення контракту зовнішньої служби «двійником».

Розв’язанням цієї проблеми є розробка сценаріїв тестування контрактів взаємодії (англ. contract test), що виконуються з певною періодичністю у системі неперервної інтеграції та виконують запити до справжніх зовнішніх компонент аби впевнитись, що їх контракт не змінився. Будь-які дефекти, що виникають під час виконання цих сценаріїв означають потребу в оновленні сценаріїв сервісного тестування для врахування змін і повинні зупинити інтеграцію створених частин програмного забезпечення до розв’язання проблем. Недоліком такого підходу є те, що сценарії виконуються періодично (найчастіше раз на день), а отже якщо зовнішня система часто змінюється, то такий підхід може бути недієвим.

Іншим підходом до розв’язання поставленої проблеми може бути синхронізація метаданих опису контрактів між системами та використання шаблону розробки «Контракт, орієнтований на споживача» (англ. consumer-driven contracts) [13]. Так, перед проведенням сервісного тестування або тестування контрактів взаємодії, система може

запросити опис контракту у зовнішньої системи мережею Інтернет і порівняти його з контрактом у сценарії тестування. Якщо контракт не змінився, то тести виконуються із використанням «двійників» у звичайному режимі. Проте у випадку зміни контракту, сценарії тестування можуть вважатися або застарілими й потребувати змін, або виконуватись з використанням справжніх викликів до зовнішніх систем. Додатково можливо проводити тестування контрактів взаємодії на стороні зовнішньої системи аби стороння команда розробників була впевнена у цілісності інтеграції їхнього компоненту з рештою компонент системи.

Вибір того чи іншого підходу обумовлений лише конкретними вимогами до тестування, а також архітектурою програмного застосунку. Останній підхід вимагає особливої структури компонентів системи, коли кожен з них може надавати опис свого прикладного програмного інтерфейсу іншим компонентам системи, що може бути неможливим у реалізації або суттєво відрізнятись від розробленої архітектури.

1.2.2 Тестування продуктивності та швидкодії

Тестування продуктивності та швидкодії, як правило, виконується з метою визначення ефективності роботи компонента або системи, їх стабільності та чутливості реагування на певні навантаження. Але тестування швидкодії також може застосовуватись для дослідження інших характеристик якості програмного застосунку, таких як відмовостійкість, масштабованість та використання ресурсів під час роботи.

Розрізняють п’ять основних напрямів проведення тестування швидкодії:

– тестування під навантаженням – тип тестування, що проводиться для оцінки можливості стабільної роботи системи під певним навантаженням (наприклад, максимальною очікуваною кількістю користувачів системи) [3];

– тестування витривалості – тестування, що проводиться для визначення межі відмовостійкості системи та ступеню навантаження, що система може витримати протягом певної кількість часу;

– тестування конфігурації системи – тип тестування для визначення оптимальної апаратної та програмної конфігурації системи для забезпечення відмовостійкості під навантаженням [3];

– стрес-тестування – тип тестування, що проводиться для оцінки системи або її компонентів в межах або поза межами передбачуваних навантажень, з можливістю часткової відмови компонентів системи [3];

– тестування піковим навантаженням – тип тестування для визначення здатності системи до відновлення після раптових пікових навантажень, а також часу, який необхідний системі для повернення до стабільного стану [3].

Деякі з наведених методів тестування швидкодії можуть бути легко автоматизовані за допомогою спеціалізованих інструментів (наприклад, тестування під навантаженням), тоді як інші потребують спеціального обладнання та інфраструктури, що значно ускладнює створення автоматизованих сценаріїв тестування та збільшує витрати на тестування.

1.2.3 Тестування безпеки

Програмне забезпечення, особливо таке, що приносить певну цінність або зберігає важливі дані користувачів, дуже часто є мішенню до всіляких типів атак. Комп’ютерні злодії, хакери, шахраї та навіть службовці можуть стати ворогами програмного застосунку і можуть спробувати як заподіяти шкоди системі, так і намагатись вкрасти цінну інформацію.

Тестування безпеки програмного забезпечення – це процес виявлення недоліків у механізмах захисту інформації системи за допомогою знань про можливі типи атак та вразливостей, що можуть призвести до відмови системи в обслуговуванні або до втрати контролю за інформацією. Завданням розробників програмного забезпечення, а також інженерів з безпеки є створення надійних засобів опору можливим атакам, а також попередження атак шляхом виявлення недоліків системи безпеки.

Розрізняють п’ять основних сфер забезпечення безпеки програмних продуктів, а саме:

– мережева безпека: передбачає пошук вразливостей мережевої інфраструктури;

– безпека системного програмного забезпечення: передбачає пошук вразливостей операційної системи, баз даних та іншого програмного забезпечення, від якого залежить програмний застосунок;

– захист клієнтського програмного забезпечення: передбачає створення механізмів контролю та моніторингу роботи програмного забезпечення клієнта (браузерів, поштових клієнтів тощо), наприклад, із застосуванням антивірусів;

– захист на рівні інтерфейсу користувача: передбачає створення механізмів валідації користувацького вводу на стороні клієнта або інтерфейсу програмного забезпечення;

– захист програмного застосунку на рівні серверу: передбачає розробку механізмів відмовостійкості у програмному застосунку для запобігання вторгненню у роботу програмного забезпечення.

Більшість з перелічених методів тестування можуть бути автоматизовані за допомогою спеціальних інструментів тестування, таких як OWASP Zed Attack Proxy [14], а також з використанням баз даних відомих вразливостей безпеки, таких як «Національна база даних вразливостей» [15]. Проте автоматизація не дає гарантію повної безпеки програмного застосунку, а тому такий тип тестування потребує обов’язкової участі інженерів з безпеки та тестування для заподіяння надійності системи.

1.2.4 Тестування локалізації

Тестування локалізації – це процес тестування перекладів, інтернаціоналізації, правопису і форматів тексту програмного застосунку, супровідної документації, а також відповідних систем або їх компонентів. Такий тип тестування дозволяє перевірити якість

адаптованості програмного продукту для певної цільової аудиторії відповідно до її культурних особливостей.

Автоматизація тестування локалізації може виконуватись різними методами, проте найпоширенішим є метод, що схожий на тестування користувацького інтерфейсу. Програмний застосунок запускається в середовищі тестування для кожної з підтримуваних мов. Далі фреймворком тестування виконується або знімок екрану користувача і його подальша перевірка з попередніми версіями, або текст компонентів інтерфейсу порівнюється з вихідними даними перекладів та локалізації.

Іншою проблемою тестування локалізації є перевірка адаптованості інтерфейсу користувача до мов з правописом справа наліво. Елементи інтерфейсу користувача для таких мов повинні бути дзеркально відображені по горизонталі. Автоматизація сценаріїв тестування таких випадків є трудомісткою задачею, адже фактично об’єктом тестування є зовсім інший інтерфейс. Досі не існує методів для ефективного проведення такого роду автоматизованого тестування, а тому найчастіше тестування локалізації виконується в ручному режимі.

1.3 Архітектура фреймворків автоматизованого тестування

Фреймворки автоматизованого тестування є невіддільною частиною будь-якого сучасного проєкту. Підвищення ефективності тестування, що реалізується за допомогою фреймворків автоматизації, перетворюється на підвищення якості програмного продукту і суттєво знижує витрати на забезпечення його контролю. При застосуванні засобів для автоматизації тестування, а також під час розробки власних фреймворків тестування, особлива увага повинна приділятись архітектурі і проєктуванню застосунку автоматизації, оскільки наявність будь-яких дефектів у такому програмному забезпечені буде означати наявність критичних вразливостей для процесу тестування, що може призвести до виникнення псевдопозитивних результатів, зниженню якості програмного забезпечення та зростанню витрат на його обслуговування.

Фреймворк автоматизованого тестування – це платформа, що спроєктований шляхом інтеграції різних апаратних та програмних ресурсів, інструментів та служб підтримки якості програмного забезпечення, що пов’язані певною архітектурою [3]. Таким чином фреймворк тестування надає всі необхідні інструменти та правила для опису й організації сценаріїв тестування, їх виконання та порівняння результатів тестування з еталонами або очікуваними результатами. Більш того, він поєднує у своїй архітектурі практики та підходи до проведення автоматизованого тестування, що допомагає початківцям почати створення сценаріїв тестування.

Цілями застосування фреймворку автоматизованого застосування є:

– підвищення ефективності та зменшення складності розробки сценаріїв автоматизованого тестування;

– повторне використання сценаріїв для проведення регресивного тестування системи;

– забезпечення структурованої методології розробки для підтримки цілісності дизайну системи та зменшення залежності від випадкових дефектів;

– забезпечення надійних інструментів виявлення дефектів та аналізу першопричин їх виникнення з мінімальним втручанням людини у процес тестування;

– зменшення залежності від експертів бізнес-домену шляхом автоматизованого реагування на зміни умов тестування системи та середовища тестування;

– ефективне використання ресурсів для проведення тестування складних систем, а також забезпечення автоматизованого неперервного контролю за процесом тестування.

Завданням створення архітектури фреймворку автоматизованого тестування є проєктування такої системи автоматизації, що здатна бути одночасно структурованою і зрозумілою, але гнучкою до змін у мовах програмування, технологіях та підходах до проведення тестування. Проблемою розробки такої архітектури є вибір оптимальної

комбінації підходів до тестування, методологій та інструментів автоматизованого тестування. Розробка гнучкої архітектури на початкових етапах проєктування є найважливішим кроком, оскільки впливає на успішність подальшої розробки й диференціює необхідні витрати не тільки на проєктування фреймворку, а й на його подальше застосування у виробничому середовищі.

Архітектуру фреймворків автоматизованого тестування з точки зору підходів до опису структури сценаріїв тестування можна умовно поділити на п’ять типів (рисунок 1.3), кожна з яких має свої переваги та недоліки: лінійні сценарії тестування, сценарії тестування керовані даними, сценарії тестування керовані ключовими словами, сценарії тестування керовані поведінкою та гібридні сценарії тестування. Вибір конкретної архітектури залежить від потреб бізнес-домену, а саме тому серед програмного забезпечення можливо спостерігати широке різноманіття фреймворків автоматизованого тестування.

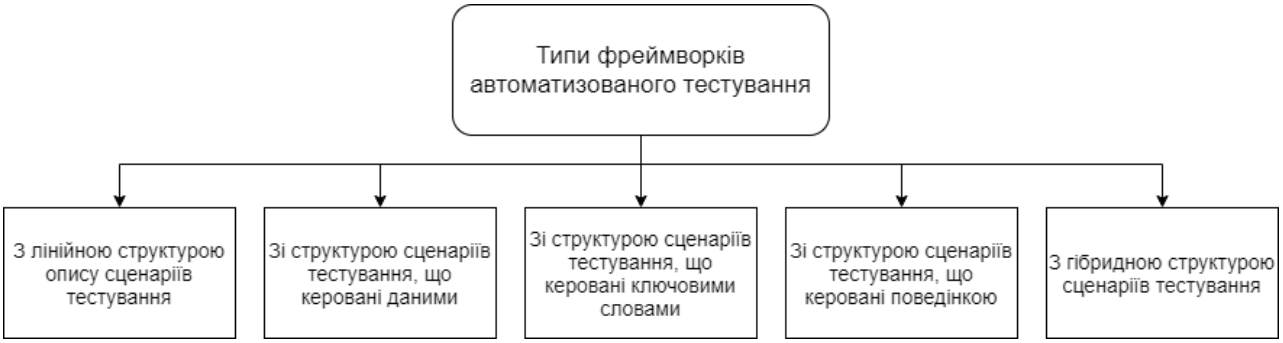


Рисунок 1.3 – Типи фреймворків автоматизованого тестування за принципом опису сценаріїв тестування

Фреймворк з архітектурою лінійних сценаріїв тестування є базовим засобом автоматизованого тестування з підходом тестування «Запис та Відтворення». Сценарії тестування такого фреймворку являють собою процедурний програмний код, що описує послідовність дій для відтворення кроків процесу тестування. Розробка сценаріїв не потребує особливих навичок програмування, а завдяки лінійності сценарії тестування є простими для розуміння. Найчастіше такі фреймворки застосовують для невеликих проєктів або в проєктах, де можливо автоматизувати генерацію сценаріїв тестування

методом послідовного запису кроків тестування. Перевагою такого типу фреймворків є простота налаштування процесу тестування, проте переважно недоліком є неможливість повторного використання сценаріїв тестування.

Фреймворк з архітектурою сценаріїв тестування, що керовані даними, орієнтований на розміщення даних для тестування поза межею сценарію тестування, що дозволяє повторно використовувати сценарії передаючи в них різні набори даних. Набір даних для тестування при такому підході розміщується у файловій системі або базах даних. Відділення даних від кроків сценарію тестування дозволяє значним чином зменшити кількість сценаріїв тестування та надає гнучкість до зміни сценарію шляхом оновлення його даних. Проте такий підхід не завжди є доцільним, адже система може потребувати наявності унікальних варіантів тестування функціональності, що при такому підході призведе до породження великої кількості сценаріїв тестування та даних для них. Іншим суттєвим недоліком такого підходу є ускладненість розуміння процесу тестування, а також потреба в особливих навичках програмування для створення такого типу сценаріїв тестування.

Фреймворк з архітектурою сценаріїв тестування, що керовані ключовими словами, орієнтований на пов’язання програмного коду системи з певними ключовими словами, що застосовуються у сценаріях тестування для виконання подій у системі. Аналогічно до підходу створення сценаріїв, що керовані даними, в сценаріях такого типу логіка сценарію, його дані та ключові слова розділені між собою та зберігаються на зовнішніх ресурсах. Послідовність ключових слів переважно записується у вигляді таблиці зіставлень подій у системі та ключових слів і описує кроки сценарію тестування. Після створення таблиці зіставлень, створюється відповідний програмний код, що відповідає створеним ключовим словам, і який буде виконуватись при виконанні сценарію тестування. Перевагою підходу до розробки сценаріїв, що керовані ключовими словами, є можливість повторного використання ключових слів у різних сценаріях тестування, але недоліком є обмежена сфера застосування цього підходу. Найчастіше

він використовується для тестування користувацького інтерфейсу, але його застосування для модульного тестування буде недоцільним внаслідок значної трудомісткості.

Фреймворк з архітектурою сценаріїв тестування, що керовані поведінкою, є найбільш популярним підходом, що орієнтований на опис причинно-наслідкових зв’язків системи. Метою такого типу фреймворків є надання спільного набору інструментів як команді розробників та інженерів з тестування, так і менеджменту компанії. Здебільшого це можливо завдяки використанню зовнішніх мов опису сценаріїв тестування, що схожі на людську мову, і які згодом обробляються фреймворком тестування для створення програмного коду кінцевих сценаріїв тестування. Найбільшим недоліком такого підходу є потреба впевнених навичок програмування в інженерів з тестування, а також створення додаткових сценаріїв тестування мовою програмного коду.

Гібридні фреймворки автоматизованого тестування поєднують у собі наявні архітектуру та підходи до опису сценаріїв тестування, що найбільш вдало зображають процеси тестування певного бізнес-домену. Створення таких фреймворків є трудомісткою задачею, проте вони найліпшим чином розв’язують задачі тестування. Значна гнучкість їх архітектури дозволяє налаштовувати фреймворк під конкретні вимоги, обирати інструменти та підходи до тестування, а також розширяти його функціональність за допомогою системи сторонніх модулів.

Спільним для будь-яких фреймворків автоматизованого тестування є процес роботи фреймворку, коли сценарії тестування зчитуються з певного ресурсу, корки сценаріїв виконуються в певній послідовності з використанням додаткової інформації, а фреймворк управляє програмним застосунком і перевіряє результати виконання сценаріїв тестування. На рисунку 1.4 зображено загальну архітектуру фреймворку автоматизованого тестування.

Рисунок 1.4 – Загальна архітектура фреймворку автоматизованого тестування

Таким чином, більшість типів фреймворків автоматизованого тестування завдяки гнучкості своєї архітектури охоплюють постійні зміни у проєктах з метою мінімізації ручного та автоматизації регресивного тестування. Шляхом застосування декількох методів тестування та автоматизованого аналізу результатів вдається в більшості випадків однозначно визначити першопричину дефекту та надати вичерпну інформацію для його усунення.

1.4 Засоби для автоматизованого тестування програмної платформи Node.js

За даними Similar Tech [16] на початок 2020 року понад 90 000 унікальних бізнес-доменів застосовують програмну платформу Node.js у свої діяльності. Для забезпечення попиту та розв’язання бізнес-задач, інфраструктура платформи Node.js та її менеджер програмних застосунків пропонують широкий вибір інструментів та фреймворків, в тому числі для проведення автоматизованого тестування. Більшість з них фокусуються на автоматизації лише певних рівнів тестування, тоді як інші пропонують комплексне рішення для проведення тестування широкого профілю. Відповідно до щорічного звіту опитування більш ніж ста тисяч розробників програмного забезпечення «The State of JavaScript» [17], найпопулярнішими засобами для автоматизованого тестування мовою програмування JavaScript у 2019 році стали: Jest, Cypress, Mocha, Jasmine, Ava та Cucumber. Всі вони тим чи іншим чином схожі між собою, розв’язують певні задачі

автоматизації тестування, але пропонують різний синтаксис опису сценаріїв тестування, а також підходи та інструменти його проведення.

Програмний засіб Jest, що є розробкою компанії Facebook з відкритим програмним кодом, є багатофункціональним фреймворком тестування та орієнтований на спрощення написання сценаріїв тестування. Він підтримує роботу з багатьма типами проєктів, що можуть бути написані на різноманітних діалектах мови програмування JavaScript.

До переваг фреймворку тестування Jest можна віднести:

– детальну та зрозумілу документацію;

– простоту в налаштуванні та гнучкі можливості конфігурації;

– виконання сценаріїв тестування у паралельному режимі, при якому в першу чергу виконуються тести, що були попередньо невдалими, а також тести, виконання яких заплановано як довготривале;

– ізольованість сценаріїв внаслідок їх виконання в окремих процесах;

– надання унікальних інструментів тестування на кшталт зліпків стану системи, збору статистик покриття коду тестами тощо;

– підтримку проєктів, що використовують фреймворк React;

– надання середовища браузерного тестування.

До недоліків фреймворку тестування Jest можна віднести:

– розміщення функціональності фреймворку у глобальних змінних, що впливає на програмний застосунок;

– довгу ініціалізацію системи, що впливає на швидкість тестування в методології розробки, керованої тестуванням;

– потребу навичок програмування для створення сценаріїв тестування;

– потребує навичок тестування програмного забезпечення та розуміння термінології;

– відсутність підтримки інтеграції сторонніх програмних застосунків;

– відсутність підтримки стандарту ECMAScript Modules;

– проблеми зі створенням та налаштуванням «двійників» під час тестування;

– для проведення тестування користувацького інтерфейсу потребує встановлення сторонніх бібліотек.

Сценарії тестування, що розроблені за допомогою фреймворку Jest, являють собою програмний код, який застосовує глобальні функції фреймворку для опису етапів тестування. На рисунку 1.5 зображено приклад сценарію тестування із застосуванням фреймворку Jest. Недоліком синтаксису таких сценаріїв можна вважати хаотичність організації даних, що описують головну інформацію про сценарій.

```
'use strict';

jest.mock('fs');

describe('listFilesInDirectorySync', () => {
  const MOCK_FILE_INFO = {
    '/path/to/file1.js': 'console.log("file1 contents");',
    '/path/to/file2.txt': 'file2 contents',
  };

  beforeEach(() => {
    // Set up some mocked out file info before each test
    require('fs').__setMockFiles(MOCK_FILE_INFO);
  });

  it('includes all files in the directory in the summary', () => {
    const FileSummarizer = require('../FileSummarizer');
    const fileSummary = FileSummarizer.summarizeFilesInDirectorySync(
      '/path/to',
    );

    expect(fileSummary.length).toBe(2);
  });
});
```

Рисунок 1.5 – Приклад сценарію тестування із застосуванням фреймворку Jest

Попри недоліки фреймворку Jest, він є багатофункціональним засобом автоматизованого тестування, що підходить для потреб багатьох проєктів. Для проведення модульного та інтеграційного тестування не потребує встановлення сторонніх бібліотек. Завдяки наявності декількох середовищ тестування надає можливість виконання сценаріїв, що розроблені для виконання у браузерах.

Програмний засіб Mocha є фреймворком автоматизованого тестування з відкритим програмним кодом. Особливістю цього фреймворку є його мінімалістична архітектура, що дозволяє виконувати сценарії як у середовищі програмної платформи Node.js, так і у браузері.

До переваг фреймворку тестування Mocha можна віднести:

– синхронність звітування про знайдені недоліки, що допомагає точно ідентифікувати невдалі сценарії;

– можливість розширення функціоналу сторонніми модулями, а також наявність великої бібліотеки плагінів;

– підтримку асинхронного тестування;

– орієнтованість синтаксису сценаріїв тестування на методологію розробки, що керована поведінкою.

До недоліків фреймворку тестування Mocha можна віднести:

– розміщення функціональності фреймворку у глобальних змінних, що впливає на програмний застосунок;

– послідовність виконання сценаріїв тестування внаслідок архітектури фреймворку;

– відсутність підтримки стандарту ECMAScript Modules;

– потребу встановлення додаткового програмного забезпечення для проведення тестування (бібліотек перевірки результатів, створення «двійників» тощо).

На рисунку 1.6 зображений приклад синтаксису сценарію тестування, що розроблений за допомогою фреймворку Mocha. Особливістю цього синтаксису є надзвичайна схожість з синтаксисом сценаріїв тестування фреймворку Jest, а отже синтаксис має ті ж самі недоліки. Однак перевагою прикладного програмного інтерфейсу фреймворку Mocha є можливість зміні синтаксису опису сценаріїв тестування шляхом застосування сторонніх бібліотек.

```
const chai = require("chai");
const sinon = require("sinon");
const expect = chai.expect;
const faker = require("faker");
const { UserModel } = require("../database");
const UserRepository = require("./user.repository");

describe("UserRepository", () => {
  const stubValue = {
    id: faker.random.uuid(),
    name: faker.name.findName(),
    email: faker.internet.email(),
    createdAt: faker.date.past(),
    updatedAt: faker.date.past()
  };

  describe("create", () => {
    const stub = sinon.stub(UserModel, "create").returns(stubValue);
    const userRepository = new UserRepository();

    it("should add a new user to the db", async () => {
      const user = await userRepository.create(stubValue.name, stubValue.email);
      expect(stub.calledOnce).to.be.true;
      expect(user.id).to.equal(stubValue.id);
      expect(user.name).to.equal(stubValue.name);
      expect(user.email).to.equal(stubValue.email);
      expect(user.createdAt).to.equal(stubValue.createdAt);
      expect(user.updatedAt).to.equal(stubValue.updatedAt);
    });
  });
});
```

Рисунок 1.6 – Приклад сценарію тестування із застосуванням фреймворку Mocha

Попри переваги та гнучкість фреймворку автоматизованого тестування Mocha, в ньому бракує вбудованих засобів тестування, а потреба в додатковій конфігурації та встановлення додаткових бібліотек ускладнює початкову роботу з фреймворком.

Програмний засіб Jasmine є одним з найстаріших фреймворків автоматизованого тестування програмної платформи Node.js. Він орієнтований на виконання сценаріїв тестування у будь-якому середовищі, де можливе виконання програмного коду мовою програмування JavaScript, що робить його універсальним рішенням для багатьох проєктів.

До переваг фреймворку тестування Jasmine можна віднести:

– детальну документацію та велику спільноту розробників, що використовують це фреймворк протягом багатьох років;

– простоту налаштування завдяки вбудованим інтерактивним інструментам конфігурування;

– високий ступінь надійності внаслідок довгої історії його існування;

– орієнтованість синтаксису сценаріїв тестування на методологію розробки, що керована поведінкою;

– наявність вбудованих інструментів для проведення модульного та інтеграційного тестування;

– підтримку декількох мов програмування.

До недоліків фреймворку тестування Jasmine можна віднести:

– відсутність гнучкості до розширення синтаксису опису сценаріїв тестування;

– розміщення функціональності фреймворку у глобальних змінних, що впливає на програмний застосунок;

– відсутність підтримки інтеграції сторонніх програмних застосунків;

– незрозумілість повідомлень про знайдені недоліки;

– проблеми тестування асинхронного коду;

– відсутність підтримки стандарту ECMAScript Modules;

– обмеженість у виборі файлової структури проєкту.

Синтаксис опису сценаріїв тестування із використанням фреймворку Jasmine є ідентичним до описаних вище фреймворків. Різницею як і у минулих випадках є тільки функціональність, що надається фреймворком під час тестування. А отже синтаксис має ті самі недоліки, що і фреймворки Jest та Mocha.

Програмний засіб Ava є інструментом виконання та валідації сценаріїв тестування з відкритим програмним кодом. Особливістю цього застосунку є орієнтованість на мінімалізм та висока швидкодія в порівнянні з фреймворками Jest, Mocha та Jasmine.

До переваг інструменту тестування Ava можна віднести:

– виконання сценаріїв тестування в паралельному режимі;
– підтримку стандарту ECMAScript Modules;
– підтримку тестування асинхронного коду;
– орієнтованість на створення атомарних сценаріїв тестування;
– підтримку діалектів мови програмування JavaScript;
– ізольованість середовища виконання сценаріїв тестування;
– відсутність впливу на глобальні змінні програмного застосунку;
– надання інструменту тестування за допомогою зліпків стану системи;
– детальний опис знайдених дефектів під час тестування.

До недоліків інструменту тестування Ava можна віднести:

– відсутність детальної документації та прикладів;
– велика кількість дефектів інструменту, що впливають на якість тестування;
– обмеженість синтаксису опису сценаріїв тестування та відсутність гнучкості до його розширення;
– відсутність підтримки інтеграції сторонніх програмних застосунків;
– необхідність встановлення сторонніх бібліотек для створення «двійників».

Синтаксис опису сценаріїв тестування із використанням інструменту виконання сценаріїв тестування Ava, хоч і є схожим на синтаксис фреймворків Jest та Mocha, проте є більш мінімалістичним і легким до читання. Так само як і в інших фреймворках мова опису являє собою програмний код, що виконується в ізольованому середовищі. На рисунку 1.7 зображений приклад синтаксису сценарію тестування, що розроблений за допомогою інструменту тестування Ava.

```
1  import test from 'ava';
2
3  test('foo', t ⇒ {
4      t.pass();
5  });
6
7  test('bar', async t ⇒ {
8      const bar = Promise.resolve('bar');
9      t.is(await bar, 'bar');
10 });
```

Рисунок 1.7 – Приклад сценарію тестування із застосуванням інструменту Ava

Програмний засіб Cucumber є фреймворком орієнтованим на розробку, що керована поведінкою. Найголовнішою особливістю даного фреймворку є особлива предметно-орієнтована мова опису сценаріїв тестування – Gherkin. Простий синтаксис цієї мови націлений на одночасну співпрацю програмістів, інженерів з контролю якості програмного забезпечення, а також керівництва компанії в процесі опису поведінкових вимог до програмного забезпечення. Написання сценаріїв тестування зводиться до формалізації вимог у вигляді натуральних конструкцій розмовної мови, що значним чином впливає на простоту читання та розуміння сценаріїв тестування. Внаслідок того, що мова опису сценаріїв являє собою звичайний текст, розробка сценаріїв тестування не вимагає навичок програмування для їх створення. Основною перевагою цього фреймворку є незалежність від середовища тестування: сценарії можуть виконуватись як на програмній платформі Node.js, так і у браузері. Проте, суттєвими недоліками фреймворку Cucumber є вплив на швидкодію тестування внаслідок необхідності парсингу тексту сценарію мови Gherkin та потреба у розробці програмного коду, який повинен виконувати кроки розібраного сценарію тестування предметно-орієнтованої мови Gherkin. Приклад синтаксису програмного коду сценарію тестування із застосуванням фреймворку Cucumber наведено на рисунку 1.8.

```
const { When, Then, After } = require('cucumber');
const assert = require('assert');
const { Builder, By, until } = require('selenium-webdriver');

When('we request the products list', async () => {
    this.driver = new Builder()
        .forBrowser('chrome')
        .build();

    this.driver.wait(until.elementLocated(By.tagName('h1')));
    await this.driver.get('http://localhost:4200');
});

Then('we should receive', async (dataTable) => {
    const productElements = await this.driver.findElements(By.className('product'));
    const expectations = dataTable.hashes();
    for (let i = 0; i < expectations.length; ++i) {
        const productName = await productElements[i].findElement(By.tagName('h3')).getText();
        assert.equal(productName, expectations[i].name);

        const description = await productElements[i].findElement(By.tagName('p')).getText();
        assert.equal(description, `Description: ${expectations[i].description}`);
    }
});

After(async () => {
    this.driver.close();
});
```

Рисунок 1.8 – Приклад сценарію тестування із застосуванням фреймворку Cucumber

Більшість з наявних методів опису сценаріїв автоматизованого тестування передбачають використання мови програмування JavaScript для опису сценаріїв тестування, що потребує певної кваліфікації в інженерів з контролю якості програмного забезпечення для написання таких сценаріїв або залучення програмістів до їх створення. Предметно-орієнтована мова Gherkin не вирішує цю проблему, оскільки вимагає написання додаткового програмного коду для виконання сценаріїв цієї мови.

1.5 Постановка задач

Проведений огляд методів та засобів до автоматизації тестування програмного забезпечення показав, що більшість з наявних підходів до створення сценаріїв автоматизованого тестування передбачають їх описання безпосередньо мовою програмування, яка застосовується у вихідному коді програмного застосунку, що тестується. Використання ж деякими фреймворками автоматизованого тестування програмної платформи Node.js предметно-орієнтованих мов, таких як Gherkin, для цих

цілей також не вирішує проблему, оскільки необхідним залишається створення додаткового програмного коду для опису сценарію тестування та пов’язання структур предметно-орієнтованої мови зі структурами програмного застосунку.

Тому для розв’язання перелічених проблем були сформовані наступні задачі розробки для даної магістерської роботи:

1) удосконалити методи та засоби створення сценаріїв автоматизованого тестування шляхом застосування підходів металінгвістичного програмування;

2) розробити фреймворк автоматизованого модульного тестування для програмної платформи Node.js;

3) провести дослідження ефективності запропонованих методів та засобів автоматизованого тестування в порівнянні з наявними або аналогічними рішеннями в програмній платформі Node.js.

1.6 Висновки до розділу

У даному розділі було проведено аналіз основних підходів до опису сценаріїв автоматизованого тестування, порівняльний аналіз методів проведення тестування та засобів програмної платформи Node.js, які застосовуються для автоматизованого тестування програмного забезпечення.

Було показано, що більшість з наявних засобів автоматизованого тестування передбачають написання сценаріїв тестування за допомогою мов програмування аналогічних тим, що застосовуються в вихідному коді проєкту. Такий підхід значним чином ускладнює створення сценаріїв тестування, вимагає певних навичок в інженерів з тестування, а також робить неможливим розуміння сценаріїв тестування стейкхолдерами програмного продукту.

Серед наявних методів опису сценаріїв найпростішою до написання є предметно-орієнтована мова Gherkin, адже вона не потребує навичок програмування для написання сценаріїв тестування і значно відрізняється від наявних аналогів. Проте, так само як й інші методи опису сценаріїв тестування потребує написання програмного коду базовою

мовою програмування для поєднання сценарію на мові Gherkin з програмними компонентами вихідної системи.

Таким чином, постає необхідним розробка предметно-орієнтованої мови програмування, що буде орієнтована на спрощення процесу автоматизації тестування та розв'язання проблеми взаємодії зацікавлених сторін програмного продукту.

2 МЕТОД ПОБУДОВИ СЦЕНАРІЇВ ТЕСТУВАННЯ

Виходячи з результатів огляду теоретичних положень галузі автоматизованого тестування програмного забезпечення, створення сценаріїв автоматизованого тестування найчастіше є об’єктом складнощів для інженерів з тестування, а отже підходи та методи їх програмного опису потребують удосконалення. Високорівневі мови опису сценаріїв тестування, такі як Gherkin, хоч і розв’язують поставлену задачу, але мають суттєвий недолік через потребу в написанні окремого програмного коду, що виконується фреймворком автоматизованого тестування з використанням даних сценарію високорівневої мови.

2.1 Дослідження наявних методів та засобів опису сценаріїв автоматизованого тестування

Однією з найпоширеніших мов створення сценаріїв автоматизованого тестування є предметно-орієнтована мова Gherkin. Застосування цієї мови для створення сценаріїв тестування допомагає описати поведінку програмної системи з точки зору бізнесу без необхідності детального опису конкретної програмної реалізації. Специфікація опису поведінки системи у мові Gherkin будується по наступній структурі:

1) заголовок або назва – в кон'юнктивній формі надається опис бізнес-цілі користувацької історії;

2) опис сценарію – коротко повинні бути надані відповіді на запитання: хто є зацікавленою стороною даної вимоги/користувацької історії, що входить до складу вимоги та яку цінність надає реалізація цієї вимоги бізнесу;

3) сценарії тестування – специфікація може містити будь-яку кількість сценаріїв тестування, необхідних для перевірки виконання поставлених вимог; кожен сценарій зазвичай будується за принципом: опис передумов, опис послідовності подій, що потрібно виконати у сценарії, очікувані результати тестування.

Мова Gherkin не надає будь-яких формальних правил опису сценаріїв автоматизованого тестування, але наполягає на використанні спеціального набору ключових слів для надання структури сценарію та опису всіх необхідних для тестування артефактів. Кожне слово предметно-орієнтованої мови Gherkin може бути перекладене у майже будь-яку розмовну мову, що робить цю мову універсальною для міжнародного використання. В таблиці 2.1 наведені основні ключові слова мови опису сценаріїв тестування Gherkin англійською мовою, а на рисунку 2.1 наведено приклад сценарію тестування цією мовою.

Таблиця 2.1 – Ключові слова мови опису сценаріїв тестування Gherkin

Ключове слово	Пояснення застосування
Story / Feature	Кожна вимога починається з цього ключового слова, після якого через двокрапку пишеться ім’я користувацької історії.
As a	Назва, тип або ім’я користувача системи, що пов’язаний з цією користувацькою історією.
In order to	Цілі, що переслідує користувач, або мета цієї користувацької історії.
I want to	Опис очікуваного результату реакції системи або цінності, що отримує користувач.
Scenario	Кожний сценарій тестування даної користувацької історії починається з цього ключового слова, після якого через двокрапку пишеться ім’я та мета сценарію. Якщо користувацька історія містить декілька сценаріїв тестування, то після ключового слова вказується порядковий номер сценарію.
Given	Передумова. Якщо передумов декілька, то кожна нова умова вказується з нового рядка за допомогою ключового слова «And»
When	Подія, яка розпочинає сценарій тестування. Якщо потрібно виконати складну подію, яку неможливо описати одним реченням, то деталі події описуються кожна з нового рядка за допомогою ключових слів «And» та «But».
Then	Результат, який користувач системи повинен спостерігати після виконання всіх кроків сценарію тестування. Якщо результат неможливо описати одним реченням, то деталі результату описуються кожна з нового рядка за допомогою ключових слів «And» та «But».
And	Допоміжне ключове слово, що є аналогом логічної кон’юнкції.
But	Допоміжне ключове слово, що є аналогом логічного заперечення.

```
Story: Returns go to stock

As a store owner
In order to keep track of stock
I want to add items back to stock when they're returned.

Scenario 1: Refunded items should be returned to stock
Given that a customer previously bought a black sweater from me
And I have three black sweaters in stock.
When they return the black sweater for a refund
Then I should have four black sweaters in stock.

Scenario 2: Replaced items should be returned to stock
Given that a customer previously bought a blue garment from me
And I have two blue garments in stock
And three black garments in stock.
When they return the blue garment for a replacement in black
Then I should have three blue garments in stock
And two black garments in stock.
```

Рисунок 2.1 – Приклад сценарію тестування на мові Gherkin

Предметно-орієнтована мова опису сценаріїв тестування Gherkin також підтримує коментування тексту сценарію тестування шляхом додавання хеш-знаку (#) на початку рядка в бідь-якому місці файлу. Для збільшення читабельності тексту також дозволяється довільне розміщення відступів та символів перенесення рядка.

Оскільки мова Gherkin є зовнішньою предметно-орієнтованою мовою створення сценаріїв автоматизованого тестування, тобто не є мовою програмування, то вона потребує застосування спеціального програмного забезпечення для виконання сценаріїв, а також написання програмного коду для додаткового описання сценаріїв тестування та їх пов’язання з програмним продуктом, що тестується. В мові програмування JavaScript таким програмним забезпеченням є фреймворк Cucumber, що розбирає специфікацію користувацької історії на складові частини, керуючись ключовими словами предметно-орієнтованої мови Gherkin.

2.2 Формалізація мови опису сценаріїв автоматизованого тестування

Для вирішення поставлених у розділі 1.5 задач, було прийнято рішення розробити дочірню до JavaScript мову програмування, що використовує метапрограмування для трансляції низькорівневих конструкцій мови програмування у металінгвістичні абстракції – певний словник ключових слів, фраз або синтаксису, що описує проблеми певної предметної галузі [18].

Оскільки однією з вимог до методу опису сценаріїв тестування є забезпечення легкого сприйняття мови при читанні та написанні, то для розв'язання цієї задачі було обрано використання синтаксису S-виразів (англ. s-expression) – нотація для позначення вкладених списків деревоподібної структури, що була вперше винайдена Джоном Маккартні [19] та знайшла широке застосування у таких мовах програмування як Lisp, Scheme та WebAssembly. Синтаксис S-виразів являє собою групу атомів, що об'єднані у круглі дужки у форматі:

$$(x\ y\ z) \equiv \Big(x\ .\ \big(y\ .\ (z\ .\ NIL)\big)\Big), \tag{1}$$

де NIL означає термінальний символ кінця списку. У контексті мови програмування JavaScript та її специфікації ECMAScript [20], синтаксис S-виразів може бути представлений у вигляді вкладених списків, що починаються з ключового слова, слідом за яким у круглих дужках через кому надаються необхідні аргументи.

Метою використання S-виразів є програмний опис структури сценаріїв тестування та тестових випадків. Згідно зі стандартом тестування та опису сценаріїв [3, 21], сценарій тестування повинен включати такі поля: унікальний ідентифікатор сценарію, пріоритет виконання, назва, опис сценарію, посилання на об'єкт тестування, передумови до кроків сценарію, послідовність кроків сценарію тестування, дані для виконання сценарію, постумови та очікувані результати. Відповідно, для забезпечення

виконання стандартів тестування, формальна мова повинна надавати синтаксис або структури даних, що можуть описувати поля сценарію тестування. Завдяки синтаксису S-виразів можливо реалізувати стандартний формат опису сценаріїв тестування та створити формальну мову, що максимально наближена до людської мови, але яка одночасно являє собою мову програмування.

Оскільки мова програмування JavaScript є контекстно-вільною формальною мовою [20], то, відповідно, і розроблювана формальна мова наслідує і розширює формалізацію її граматики. За ієрархією Чомскі-Шутценберґера [22], така граматика належить до другого типу формальних граматик і може бути описана четвіркою (2), що складається з наступних складових: N – скінченна множина нетермінальних символів, Σ – скінченна множина термінальних символів, P – скінченна множина правил виводу речень формальної мови та S – початковий символ, що завжди має бути нетерміналом.

$$G = (N, \Sigma, P, S) \tag{2}$$

Термінальним алфавітом розроблюваної формальної мови виступає множина символів таблиці Юнікод 8-бітного кодування, що містить коди всіх символів, які підтримуються мовою програмування JavaScript. Однак, для опису сценаріїв тестування даною формальною мовою використовується обмежена підмножина символів стандартної таблиці ASCII, що входить у формат Юнікод, і охоплює великі та малі літери латинської абетки, цифри, деякі спеціальні та управляючі символи. Ця підмножина наведена у таблиці 2.1 у вигляді частини стандартної таблиці ASCII.

Таблиця 2.1 – Термінальний алфавіт формальної мови опису сценаріїв тестування у представлені підмножини стандартної таблиці ASCII

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL									HT	LF	VT		CR		
1																
2		!	“	#	$	%	&	‘	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4		A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	

Алфавітом нетермінальних символів даної формальної мови виступає множина об’єктів мови JavaScript, а також об’єктів формальної мови: команд, ключових слів, формул тощо. Для реалізації вимог модульного тестування, сценарії тестування повинні підтримувати такі типи примітивів та об’єктів мови програмування JavaScript:

– Undefined – невизначений тип або порожня множина;
– Boolean – логічний тип;
– Number – 64-бітне число подвійної точності формату IEEE 754;
– String – послідовність символів;
– BigInt – число у форматі довгої арифметики;
– Symbol – унікальний ідентифікатор з необов’язковим описом;
– Null – особливий примітив невизначеності об’єктного формату;
– Object – структура даних для зберігання примітивів або інших об’єктів;
– Function – фрагмент коду, що може бути викликаний іншим кодом або сам собою, або змінна, що належить до функцій.

Для забезпечення можливості розділення сценарію тестування та даних для його виконання, формальною мовою повинна бути реалізована підтримка створення змінних. Внаслідок того, що мова програмування JavaScript є мовою з динамічною типізацією, формат створення змінних у сценарії являє собою формулу еквівалентності (3), де лівою

частиною є строкова назва змінної, а правою – примітивне значення або об’єкт мови програмування JavaScript. Вираз створення змінної (3) обов’язково повинен закінчуватись символом «крапка з комою».

$$variable\_name = expression; \tag{3}$$

Відповідно до стандартів опису сценаріїв тестування [21], формальна мова має також описувати нетермінальні символи, що необхідні для опису сценарію тестування, а саме: CHAIN, KEYWORD, PARAMETER, PROGRAM, SCENARIO та SCENARIO_BODY.

Таким чином, повний алфавіт нетермінальних символів розробленої формальної мови являє собою множину (4):

$$\begin{gathered} N = \{\ \text{ARRAY, ARRAY\_BODY, BIGINT, BINARY, BINARY\_CHAIN,} \\ \text{BOOLEAN, CHAIN, CHARACTER, CHARACTER\_SEQUENCE, COMMENT,} \\ \text{DECIMAL, DIGIT, FLOAT, FLOAT\_RIGHT, HEXADECIMAL,} \\ \text{HEXADECIMAL\_CHAR, HEXADECIMAL\_DIGIT, KEYWORD,} \\ \text{LINE\_BREAK, LITERAL, MULTILINE\_COMMENT\_BODY, NULL, NUMBER,} \\ \text{OBJECT, OBJECT\_EXPRESSION, OCTAL, OCTAL\_DIGIT, PARAMETER,} \\ \text{PROGRAM, SCENARIO, SCENARIO\_BODY, STRING, UNDEFINED,} \\ \text{VARIABLE, VARIABLE\_NAME, VARIABLE\_STATEMENT }\}. \end{gathered} \tag{4}$$

Правила для виводу речень даної формальної мови, можуть бути описані у вигляді формули (5), де лівою частиною формули є нетермінальний символ, а правою частиною – будь-яка послідовність термінальних та нетермінальних символів, а зірочкою позначається оператор Кліні – унарна операція об’єднання множини символів та порожнього рядка λ.

$$\forall(w_1 \to w_2) \in P : (w_1 \in N) \cap (w_2 \ \in \ (N \ \cup \ \Sigma)^*) \tag{5}$$

У таблиці 2.2 наведено перечень всіх правил виведення речень розробленої формальної мови та відповідних пояснень до цих правил.

Таблиця 2.2 – Правила виведення речень формальної мови опису сценаріїв тестування

Правило або група правил	Опис
LINE_BREAK ::= \n LINE_BREAK ::= \r \n	Розривом рядка є термінальний символ \n або послідовність термінальних символів \r \n.
NULL ::= n u l l	Невизначеним об'єктом мови програмування JavaScript є послідовність термінальних символів n, u та подвійної літери l.
UNDEFINED ::= u n d e f i n e d	Невизначеним літералом мови програмування JavaScript є послідовність термінальних символів u, n, d, e, f, i, n, e та d.
BOOLEAN ::= t r u e BOOLEAN ::= f a l s e	Логічним літералом мови програмування JavaScript є послідовність термінальних символів t, r, u, e або f, a, l, s, e.
DIGIT ::= 1|2|3|4|5|6|7|8|9	Цифрою є один символ від 1 до 9.
DECIMAL ::= DIGIT DECIMAL ::= DECIMAL DIGIT DECIMAL ::= DECIMAL 0 DECIMAL ::= - DECIMAL DECIMAL ::= + DECIMAL	Цілим числом є або одна цифра, або послідовність цифр, що може включати в себе цифру 0, а також починатись з символу плюс або мінус.
FLOAT ::= DECIMAL . FLOAT_RIGHT FLOAT_RIGHT ::= DIGIT FLOAT_RIGHT ::= FLOAT_RIGHT DIGIT FLOAT_RIGHT ::= FLOAT_RIGHT 0 FLOAT_RIGHT ::= 0 FLOAT_RIGHT	Числом з плаваючою крапкою є ціле число, після якого ставиться крапка і слідує послідовність з цифр, що може починатись з нуля.
BINARY_CHAIN ::= 0 BINARY_CHAIN ::= 1 BINARY_CHAIN ::= BINARY_CHAIN BINARY_CHAIN BINARY ::= 0 b BINARY_CHAIN BINARY ::= 0 B BINARY_CHAIN	Бінарним числом є будь-яка послідовність символів 0 або 1, що починається з послідовності терміналів 0b або 0B.

Продовження таблиці 2.2

OCTAL_DIGIT ::= 0|1|2|3|4|5|6|7 OCTAL_DIGIT ::= OCTAL_DIGIT OCTAL_DIGIT OCTAL ::= 0 o OCTAL_DIGIT OCTAL ::= 0 O OCTAL_DIGIT	Вісімковим числом є будь-яка послідовність цифр від 0 до 7, що починається з послідовності терміналів 0o або 0O.
HEXADECIMAL_CHAR ::= A|B|C|D|E|F HEXADECIMAL_DIGIT ::= 0 HEXADECIMAL_DIGIT ::= DIGIT HEXADECIMAL_DIGIT ::= HEXADECIMAL_CHAR HEXADECIMAL_DIGIT ::= HEXADECIMAL_DIGIT HEXADECIMAL_DIGIT HEXADECIMAL ::= 0 x HEXADECIMAL_DIGIT HEXADECIMAL ::= 0 X HEXADECIMAL_DIGIT	Шістнадцятковим числом є будь-яка послідовність цифр 1-9 або символів A-F, що починаються послідовності терміналів 0x або 0X.
BIGINT ::= DECIMAL n BIGINT ::= BINARY n BIGINT ::= OCTAL n BIGINT ::= HEXADECIMAL n	Числом у форматі довгої арифметики є будь-яке десяткове ціле, бінарне, вісімкове або шістнадцяткове число, що закінчується на термінальний символ n.
NUMBER ::= DECIMAL NUMBER ::= FLOAT NUMBER ::= BINARY NUMBER ::= OCTAL NUMBER ::= HEXADECIMAL NUMBER ::= BIGINT	Типом даних «Число» мови програмування JavaScript може бути будь-яке десяткове ціле, бінарне, вісімкове, шістнадцяткове число або число у форматі довгої арифметики.
CHARACTER ::= A-z|0|DIGIT|<WS> CHARACTER_SEQUENCE ::= CHARACTER CHARACTER_SEQUENCE ::= CHARACTER_SEQUENCE CHARACTER	Послідовністю символів є один термінал або будь-яка послідовність терміналів букв A-z, цифр, пробілу, спеціальних або управляючих символів.
STRING ::= " CHARACTER_SEQUENCE " STRING ::= ' CHARACTER_SEQUENCE '	Рядком мови програмування JavaScript є будь-яка послідовність символів, що починається та закінчується терміналом “ або ‘.

Продовження таблиці 2.2

OBJECT_EXPRESSION ::= CHARACTER_SEQUENCE : LITERAL|VARIABLE OBJECT_EXPRESSION ::= STRING : LITERAL|VARIABLE	Об’єктним виразом є будь-яка послідовність символів, що розділена двокрапкою, де ліва частина є послідовністю символів або рядком, а права частина є літералом або змінною.
OBJECT ::= { } OBJECT ::= { OBJECT_EXPRESSION } OBJECT ::= { OBJECT_EXPRESSION, OBJECT_EXPRESSION }	Об’єктом мови програмування JavaScript є послідовність символів, що починається та закінчується фігурними дужками, і містить нуль або більше об’єктних виразів, що розділяються комою.
ARRAY_BODY ::= LITERAL ARRAY_BODY ::= VARIABLE ARRAY_BODY ::= ARRAY_BODY , ARRAY_BODY ARRAY ::= [ARRAY_BODY]	Масивом у мові програмування JavaScript є будь-яка послідовність літералів або змінних, що розділені комою та починаються і закінчуються термінальними символами [та] відповідно.
LITERAL ::= NULL LITERAL ::= UNDEFINED LITERAL ::= BOOLEAN LITERAL ::= NUMBER LITERAL ::= STRING LITERAL ::= OBJECT LITERAL ::= ARRAY	Літералом мови програмування JavaScript є невизначений об’єкт або невизначений літерал, логічний тип, число, рядок, об’єкт або масив.
VARIABLE ::= CHARACTER_SEQUENCE|$|_	Ім’ям змінної є бідь-яка послідовність символів, що може включати в себе знак $ та _ .
VARIABLE_STATEMENT ::= VARIABLE = LITERAL|VARIABLE ;	Виразом конструювання змінної є ім’я змінної після якого через знак дорівнює приводиться літерал або ім’я іншої змінної, а вираз закінчується крапкою з комою.
MULTILINE_COMMENT_BODY ::= CHARACTER_SEQUENCE MULTILINE_COMMENT_BODY ::= CHARACTER_SEQUENCE LINE_BREAK MULTILINE_COMMENT_BODY ::= CHARACTER_SEQUENCE LINE_BREAK MULTILINE_COMMENT_BODY	Змістом багаторядкового коментаря є будь-яка послідовність символів, що може бути розділена розривом рядка.

Продовження таблиці 2.2

COMMENT ::= // CHARACTER_SEQUENCE COMMENT ::= /* MULTILINE_COMMENT_BODY */	Коментарем є будь-яка послідовність символів, що починається з двох термінальних символів / або яка починається та закінчується термінальними символами /* та */ відповідно.
SCENARIO ::= s c e n a r i o ` CHARACTER_SEQUENCE ` (SCENARIO_BODY) ;	Сценарій тестування починається з послідовності термінальних символів s, c, e, n, a, r, i, o, `, після яких іде будь яка послідовність символів, що є назвою сценарію і яка закінчується терміналом `. Далі у круглих дужках надається тіло сценарію.
KEYWORD ::= d e s c r i p t i o n KEYWORD ::= e n t i t y KEYWORD ::= s o u r c e F i l e KEYWORD ::= d o u b l e s KEYWORD ::= f a k e F u n c t i o n KEYWORD ::= b e f o r e KEYWORD ::= a f t e r KEYWORD ::= c a l l KEYWORD ::= s a v e R e t u r n A s KEYWORD ::= c a s e s KEYWORD ::= t e s t KEYWORD ::= s t e p s KEYWORD ::= e x p e c t KEYWORD ::= p r e c o n d i t i o n KEYWORD ::= p o s t c o n d i t i o n	Ключовим словом є послідовність терміналів, що формує наступні ключові слова: description, entity, sourceFile, doubles, fakeFunction, before, after, call, saveReturnAs, cases, test, steps, expect, precondition та postcondition.
PARAMETER ::= LITERAL PARAMETER ::= VARIABLE PARAMETER ::= PARAMETER , LITERAL	Параметром може бути послідовність літералів або змінних, що розділені між собою комою.
CHAIN ::= (KEYWORD ` CHARACTER_SEQUENCE ` , PARAMETER) CHAIN ::= CHAIN CHAIN	Ланцюгом може бути одна або більше послідовностей ключових слів, параметрів та назв, що починаються та закінчуються круглими дужками.

Продовження таблиці 2.2

SCENARIO_BODY ::= KEYWORD (SCENARIO_BODY) SCENARIO_BODY ::= KEYWORD ` CHARACTER_SEQUENCE ` SCENARIO_BODY ::= KEYWORD ` CHARACTER_SEQUENCE ` (SCENARIO_BODY) SCENARIO_BODY ::= KEYWORD CHAIN SCENARIO_BODY ::= SCENARIO_BODY , SCENARIO_BODY	Тілом сценарію тестування є один або декілька наступних випадків: – ключове слово за яким іде ланцюжок; – послідовність з ключового слова, назви та тіла сценарію у круглих дужках; – послідовність з ключового слова та назви; – послідовність з ключового слова та тіла сценарію у круглих дужках.
PROGRAM ::= SCENARIO PROGRAM ::= COMMENT PROGRAM PROGRAM ::= PROGRAM COMMENT PROGRAM ::= VARIABLE_STATEMENT PROGRAM	Програмою є сценарій тестування, що може починатись з визначень змінних або коментарів, а також мати коментарі після об’явлення сценарію.

Початковим нетермінальним символом формальної мови є символ PROGRAM, який з використанням правил виводу речень формальної мови може розкладатись в опис сценарію автоматизованого тестування у форматі модифікованих S-виразів, внаслідок особливості синтаксису мови програмування JavaScript. У Додатку А приведено формалізацію створеної формальної мови у розширеній нотації Бекуса-Наура [23].

Для проміжного представлення створеної формальної мови у мовах програмування (за допомогою парсерів або шляхом застосування метапрограмування) та для реалізації можливості виконання програм, що написані цією мовою, структура програми повинна бути трансльована в абстрактне синтаксичне дерево (АСД). Оскільки синтаксис програми сформований за допомогою S-виразів, то вся програма являє собою деревоподібну структуру. В термінах комп’ютерної лінгвістики, абстрактне синтаксичне дерево охоплює результати аналізу речень формальної мови або певного ланцюга виразів, може включати дані про синтаксичне відношення між знайденими лексемами або семантичну інформацію [24]. Реалізація абстрактного синтаксичного

дерева на пряму залежить від мови програмування, в рамках якої реалізовується цей синтаксис, або від типу парсеру для розбору цього синтаксису. В загальному випадку, приклад сценарію тестування, що описаний розробленою формальною мовою, а також відповідне до нього абстрактне синтаксичне дерево представлене у додатку Б.

Програма може містити коментарі розробників, що не мають семантичного значення для сценарію програми, а отже їх представлення в абстрактному синтаксичному дереві непотрібне.

В мовах програмування, таких як JavaScript, абстрактне синтаксичне дерево може бути отримане за допомогою методів метапрограмування: наприклад, трансляції високорівневих структур даних формальної мови (S-виразів) у низькорівневі структури мови програмування (об’єкти та масиви). Представлення дерева у вигляді вкладених один в одне об’єктів є найефективнішою формою для мови програмування JavaScript, адже дозволяє зберігати зовнішні посилання на змінні та примітиви всередині ієрархії об’єктів.

Для інших мов програмування або у випадку якщо синтаксис формальної мови неможливо описати базуючись на синтаксисі вихідної мови, розроблену формальну мову можна представити у вигляді синтаксичного дерева розбору, що створюється парсиром та в якому корінним елементом є аксіома формальної мови (початковий нетермінал), проміжними елементами вершин є інші нетермінали, виведені за допомогою формальних правил, а листковими елементами виступають термінальні символи. Далі із допомогою синтаксичного дерева розбору можливо визначити лексеми мови, які в поєднанні з семантикою формальної мови утворюють логічні вирази.

Для забезпечення достатності застосування формальної мови для розв’язання задачі опису сценаріїв тестування, повинні існувати додаткові семантичні правила виразів для цієї мови, що допомагають складати логіко-достовірні вислови. Згідно розглянутому стандарту опису сценаріїв тестування [21], на рисунку 2.2 наведено структурне дерево елементів сценарію тестування. Кожна вершина являє собою елемент

сценарію тестування, кожна дочірня їй вершина показує ієрархію вкладеності елементів сценарію.

Рисунок 2.2 – Структура сценарію автоматизованого тестування

Таким чином, базуючись на семантичній структурі сценарію тестування та на правилах перетворення розробленої формальної мови, можливо створювати сценарії автоматизованого тестування, що схожі на опис людською мовою, але являються мовою програмування.

2.3 Висновки до розділу

У даному розділі запропоновано новий метод опису сценаріїв автоматизованого тестування у вигляді предметно-орієнтованої мови програмування, що є дочірньою мовою до мови JavaScript.

Розроблена формальна мова застосовує синтаксис S-виразів, що дозволяє спростити процес розробки сценаріїв автоматизованого тестування та відрізняється від аналогічних підходів до опису сценаріїв лаконічністю її сприйняття. Сценарії тестування описуються у стандартизованому вигляді шляхом застосування ключових слів формальної мови.

Оскільки розроблена предметно-орієнтована мова є мовою програмування, то на відміну від наявних аналогів вона не потребує створення додаткових програмних

артефактів для проведення автоматизованого тестування. Внаслідок використання S-виразів, розроблена предметно-орієнтована мова максимально наближена до звичайної розмовної мови, що розв'язує проблему розуміння сценаріїв тестування стейкхолдерами програмного продукту, а написання сценаріїв тестування на розробленій формальній мові не потребує навичок програмування.

Також у цьому розділі було запропоновано підхід до реалізації розробленої формальної мови у вигляді абстрактного синтаксичного дерева або у вигляді ієрархії об’єктів та масивів за допомогою металінгвістичного програмування у таких мовах як JavaScript.

3 ПРОЄКТУВАННЯ ЗАСОБУ АВТОМАТИЗАЦІЇ ТЕСТУВАННЯ

Оскільки фундаментальною основною фреймворку автоматизованого тестування є розроблена формальна мова опису сценаріїв тестування, то, відповідно, головною задачею розробки фреймворку є програмна реалізація цієї формальної мови шляхом трансляції інструкцій розробленої мови в інструкції іншої мови програмування. Для розв’язання цієї задачі було обрано мову програмування JavaScript та програмну платформу Node.js, що виступає JavaScript-оточенням, побудованим на JavaScript-рушієві Chrome V8. Обґрунтуванням такого вибору є той факт, що мова програмування JavaScript є мультипарадигмальною та надає потужні інструменти для метапрограмування. Шляхом застосування принципів метапрограмування, розроблена формальна мова у форматі S-виразів може бути представлена у JavaScript за допомогою високорівневих металінгвістичних абстракцій – певного словника ключових слів, фраз, синтаксису та структур даних, що описують предметну галузь [18].

На момент написання цієї магістерської роботи за даними індексу GitHub [25], мова програмування JavaScript є найбільш популярною мовою програмування у світі для проєктів з відкритим програмним кодом, що значним чином актуалізує причину вибору цієї мови. На додачу, результати досліджень характеристик наявних засобів автоматизації тестування для програмної платформи Node.js показали, що більшість фреймворків тестування має суттєві недоліки забезпечення підтримки новітнього стандарту програмних застосунків ECMAScript Modules [20], а також недоліки систем створення програмних «двійників» для тестування. Таким чином, програмна платформа Node.js є найліпшим вибором як цільового середовища для створення засобу автоматизації тестування.

3.1 Аналіз вимог до програмного забезпечення

Ґрунтуючись на результатах дослідження архітектури фреймворків автоматизованого тестування, а також на результатах дослідження характеристик наявних засобів тестування для програмної платформи Node.js, фреймворк для автоматизованого тестування повинен відповідати наступним характеристикам:

– надійність: фреймворк повинен гарантувати, що результати тестування є точним відображенням стану системи на момент тестування;

– ефективність: фреймворк повинен керувати ресурсами системи для забезпечення автоматизації тестування, виконувати сценарії в паралельному режимі, ефективно застосовуючи апаратні та програмні засоби;

– переносимість: фреймворк повинен легко встановлюватись в будь-якому середовищі та стеку технологій;

– масштабованість: за необхідності розширення проєкту, фреймворк повинен дозволяти зміну й перегрупування сценаріїв тестування та відповідних їм ресурсів шляхом організованої структуризації;

– повторюваність: фреймворк повинен надавати можливість багаторазового виконання групи сценаріїв тестування без необхідності внесення будь-яких змін або виконання додаткових конфігурацій;

– прозорість: результати тестування та будь-яка діяльність фреймворку повинна чітко конспектуватись у файлах логування, що доступні користувачам, а також на засобах системного введення-виведення;

– версіонування: фреймворк повинен забезпечувати облік сценаріїв тестування, будь-якої супровідної інформації, а також результатів тестування;

– гнучкість: фреймворк повинен надавати гнучку платформу тестування, що підтримує додавання нових функціональних можливостей, у тому числі мови опису сценаріїв тестування, шляхом створення зовнішніх програмних застосунків;

– сучасність: фреймворк тестування повинен застосовувати наявні сучасні можливості технологій та мов програмування.

Необов’язково але бажано, фреймворк автоматизованого тестування може відповідати наступним характеристикам:

– віддалене виконання: фреймворк надає можливість створення та завантаження завдань, виконання яких відкладене у часі або командою до виконання яких є зовнішнє повідомлення;

– відстеження історії виконання: фреймворк повинен зберігати історію виконання сценаріїв тестування; у разі невдалого попереднього запуску сценарію при наступному виконанні повинен віддати пріоритет першочерговості такому сценарію;

– відмовостійкість: у разі фатальних збоїв роботи фреймворк повинен автоматично ідентифікувати першопричину збою, визначити механізм відновлення та без додаткового втручання зреагувати належним чином;

– різноманітність: фреймворк повинен надавати доступ до вичерпної бібліотеки інструментів забезпечення автоматизованого тестування, підходів та методологій тестування;

– покриття: фреймворк у результатах тестування повинен за потреби надавати статистику покриття коду системи сценаріями тестування.

3.2 Архітектура програмного забезпечення

Виходячи з того, що середня кількість сценаріїв автоматизованого тестування у середніх за розміром проєктах на Node.js варіюється від 50 до 200, то, відповідно, виконання одного сценарію тестування та перевірка результатів його виконання має бути швидкодійною операцією, яка в тому числі ефективно використовує ресурси системи.

Програмна платформа Node.js є однопотоковим середовищем, в якому блокуючі операції (читання та запис файлів, робота з базою даних тощо) виконуються в асинхронному режимі. Node.js виконує код програми у так званому «циклі подій» та

пропонує автоматичний пул потоків для виконання завдань, що можуть займати велику кількість процесорного часу. Але оскільки пул потоків обмежений, то програмний застосунок для Node.js повинен розважливо використовувати його. Головним правилом розробки для цієї програмної платформи є те, що «Node.js надає максимальну швидкодію за умови, що у будь-який момент часу робота, пов’язана з кожним клієнтом, є мінімальною» [26].

Таким чином, для забезпечення швидкодії програмного застосунку та мінімізації часу виконання одного сценарію тестування, головною потребую для платформи Node.js є мінімізація складності операцій та їх кількості. Внаслідок використання метапрограмування для забезпечення реалізації синтаксису формальної мови, можливим є знехтування парсингом формальної мови. Проте, для трансляції інструкцій формальної мови у JavaScript-команди, необхідним є створення спрощеного абстрактного синтаксичного дерева, що описує структуру та елементи сценарію тестування, але така задача не є важкою з точки зору алгоритмізації. Інші процеси роботи фреймворку автоматизованого тестування не потребують значних ресурсів, а отже максимальної швидкодії можливо досягти завдяки рівномірному розподіленню задач по наявним потокам програмної платформи Node.js.

Ґрунтуючись на результатах аналізу вимог до програмного забезпечення та можливих архітектурних рішеннях для засобів автоматизованого тестування програмного забезпечення, було розроблено архітектуру програмного застосунку, що зображена на діаграмі компонент (рисунок 3.1).

Рисунок 3.1 – Архітектура застосунку для автоматизації тестування

Архітектурне рішення складається з наступних компонент:

– ядро фреймворку – забезпечує пов’язання всіх компонентів архітектури, програмну реалізацію формальної мови опису сценаріїв тестування, а також яка є вхідною та вихідною точкою системи;

– програмний інтерфейс командного рядка – інструменти організації програмного вводу-виводу;

– управління залежностями – компонент контролю пісочниць віртуальних машин та засіб впровадження залежностей для програмного застосунку, що тестується;

– виконавець сценаріїв тестування – компонент пошуку сценаріїв тестування, їх пріоритезації, організації у чергу та паралельного виконання;

– доповідач – компонент збору статистики виконання сценаріїв тестування та її подальшого форматування;

– створення «двійників» тестування – забезпечує підроблення компонентів програмної системи користувача або інших систем, від яких залежить програмний застосунок користувача;

– програмний контекст – бібліотека змінних, констант, посилань та , яке є програмним оточення певного компонента.

3.3 Реалізація функціоналу програмного забезпечення

Реалізація формальної мови в рамках мови програмування JavaScript являє собою створення вбудованої предметно-орієнтованої мови. Кожне ключове слово формальної мови являє собою функціональний об’єкт у мові програмування JavaScript. Так, наприклад, ключове слово «scenario» є позначеним шаблонним літералом, де позначкою є саме ключове слово «scenario», яке визначає створення об’єкту сценарію, а шаблонним літералом є рядок, який визначає назву сценарію. Оскільки позначка шаблону є функціональним об’єктом, то шляхом замикання контексту функції, вона повертає у контекст інший функціональний об’єкт, який визначає межі сценарію тестування та приймає в якості аргументів інші поля сценарію тестування.

Реалізація ключових слів «before» та «after», або інших ключових слів опису лінійної послідовності дій, представляється у вигляді застосування монад – функціональних об’єктів зі станом, які зберігають лінійну послідовність пов’язаних команд. Ці функціональні об’єкти при кожному зверненні до них зберігають аргументи, з якими був виконаний виклик до них (в даному випадку з кроками, що повинні бути виконані як передумови або постумови сценарію) та повертають замкнене посилання на самих себе. Таким чином, згодом при створенні спрощеного абстрактного синтаксичного дерева сценарію тестування, ці збережені аргументи будуть трансформовані у матрицю послідовності кроків передумов та постумов сценарію тестування.

Деякі ключові слова, наприклад «steps» та «test», представляються у вигляді функціональної композиції, що відображає ієрархію структури полів сценарію. Функціональна композиція починається з ключового слова формальної мови, слідом за яким у круглих дужках надаються аргументи – ієрархічно пов’язані між собою поля

сценарію тестування. Таким чином будується структура сценарію, а кого компоненти діляться на розділи за принципом виконання згідно зі стандартом тестування.

З точки зору синтаксису мови програмування JavaScript та з точки зору семантичного вигляду сценарію програми, синтаксис формальної мови являє собою набір ключових слів, що розміщений у глобальних змінних вихідного коду сценарію тестування. Але, якщо б ключові слова були розміщені у глобальних змінних, то їх контекст поширився б і на програмний застосунок користувача, що могло б вплинути на нього та в найгіршому випадку призвести до відмови у виконанні.

Для розв’язання цієї проблеми було застосовано віртуальну машину JavaScript-рушія Chrome V8, яка надається програмною платформою Node.js. Віртуальна машина дозволяє виконувати програмний код в ізольованому середовищі (так званих пісочницях) і точково управляти програмним контекстом виконання, що не поширюється на інші сценарії, процеси та потоки без явної на це вказівки. Програмним контекстом віртуальної машини для вихідного коду сценарію тестування є сукупність ключових слів формальної мови у вигляді функціональних об’єктів, а також необхідних компонентів для опису сценарію тестування на кшталт засобу створення «двійників». Цей програмний контекст також наслідує стандартний контекст програмної платформи Node.js, що включає в себе ключові слова мови програмування JavaScript.

Створеною формальною мовою передбачено також можливість створення змінних для їх застосування в межах сценарію тестування. У мові програмування JavaScript будь-яка змінна створюється із застосуванням ключових слів «let» або «var», а констант із застосуванням ключового слова «const». Після цих ключових слів наводиться назва змінної або константи і через символ «дорівнює» наводиться бажаний вміст змінної або константи. Оскільки синтаксис створеної формальної мови передбачає створення змінних без вказання будь-яких ключових слів перед їх об’явленням, то постала проблема реалізації такого синтаксису в рамках мови програмування JavaScript.

Програмний контекст віртуальної машини являє собою об’єкт, що може бути мутований в ході виконання сценарію програми. Такими мутаціями можуть виступати редагування змінних, перепризначення або створення нових глобальних змінних, створення констант тощо. Всі мутації відображаються у програмному контексті, який контролюється ядром фреймворку автоматизованого тестування, але за замовчуванням фреймворк не має контролю над процесами створення змінних у мові JavaScript. Проте завдяки гнучкості цієї мови та наявності потужних засобів метапрограмування, можливо організувати перехоплення подій створення змінних та забезпечення відповідного контролю. Для реалізації таких дій, мовою програмування JavaScript пропонується можливість створення проксі-об’єктів, що виступають прошарком спостереження та перехоплення стандартних подій над об’єктами. В даному випадку для розв’язання поставленої задачі необхідно обгорнути програмний контекст віртуальної машини у проксі-об’єкт та створити наступні перехоплювачі:

– отримання змінної – за допомогою рефлексії проксі-об’єкт перевіряє, чи була створена змінна і якщо вона була попередньо створена, то повертає її значення, а якщо ні, то створює змінну зі значення невизначеного об’єкта та повертає це значення;

– створення змінної – проксі-об’єкт отримує назву змінної та її бажане значення і за допомогою рефлексії створює у програмному контексті віртуальної машини цю змінну.

Таким чином, внаслідок того, що проксі-об’єкт перехоплює події створення та отримання глобальних змінних програмного контексту віртуальної машини, стає можливим реалізація синтаксису об’явлення змінних створеної формальної мови без необхідності застосування ключових слів мови програмування JavaScript.

Далі, оскільки цільовою моделлю структури програмних застосунків фреймворку є стандарт ECMAScript Modules, то, відповідно, процес завантаження та виконання вихідного коду сценарію тестування може бути розділений на наступні кроки:

– зчитування файлу вихідного коду сценарію тестування з диску та його запис у пам’ять програми;

– підготовка програмного контексту віртуальної машини і завантаження ключових слів формальної мови;

– обгортання програмного контексту віртуальної машини у проксі-об’єкт та створення перехоплювачів подій;

– створення віртуальної машини з вихідного коду сценарію тестування за допомогою створеного програмного контексту;

– поєднання необхідних залежностей та їх прив’язка до модулю сценарію тестування;

– виконання вихідного коду сценарію тестування.

Результатом виконання вихідного коду сценарію тестування є спрощене абстрактне синтаксичне дерево, яке являє собою структуру сценарію у вигляді сукупності вкладених один в одне об’єктів та масивів даних. На рисунку 3.2 зображено діаграму послідовності підготовки програмного оточення та виконання сценарію тестування фреймворком.

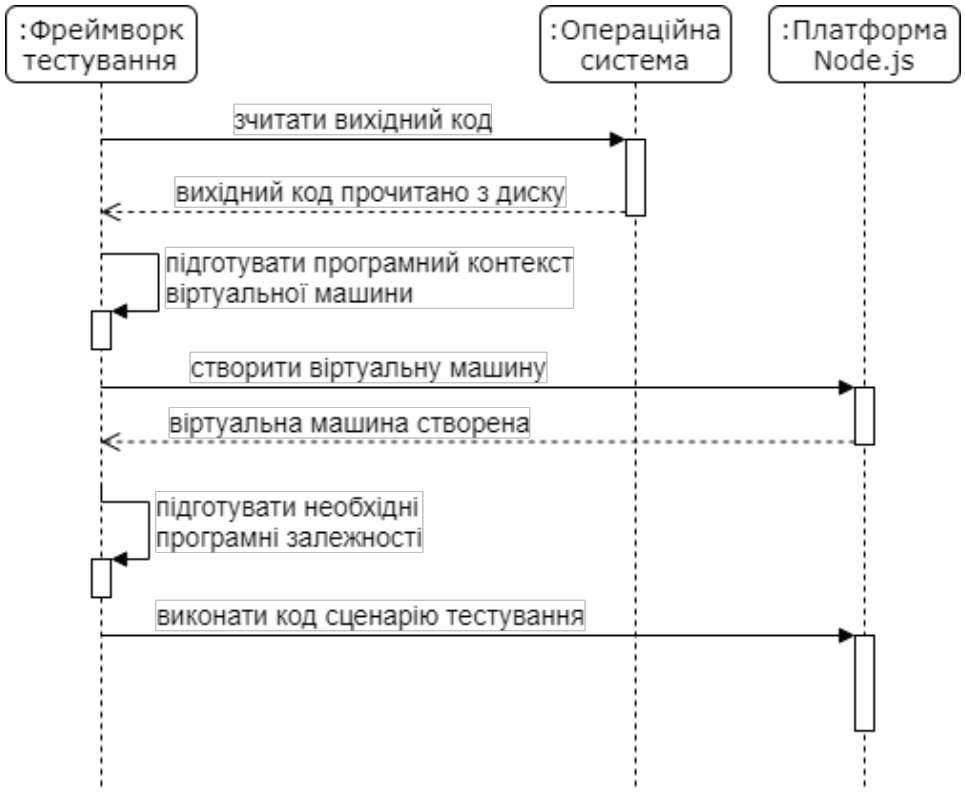


Рисунок 3.2 – Послідовність підготовки та виконання вихідного коду сценарію тестування

Наступними кроками роботи фреймворку для виконання сценарію тестування є отримання посилання зі спрощеного абстрактного синтаксичного дерева на назву файлу з вихідним кодом програмного застосунку, що необхідно протестувати. Якщо посилання не було надано у сценарії тестування, то сценарій вважається недійсним і фреймворк переходить до опрацювання наступного сценарію. В іншому випадку, посилання успішно отримується і фреймворк ініціює такий самий процес створення віртуальної машини, що був описаний вище для віртуальної машини опрацювання сценарію тестування. Суттєвою різницею створення віртуальної машини для програмного застосунку клієнта від віртуальної машини сценарію тестування є процеси створення програмного контексту та підготовки залежностей.

Оскільки однією з потреб модульного тестування є створення «двійників» компонентів системи користувача, то, відповідно, такі підроблені компоненти повинні бути створені до етапу виконання коду користувача. Для створення «двійників»

фреймворком тестування зчитується метаінформація про «двійників» зі спрощеного абстрактного синтаксичного дерева та на основі цієї метаінформації створюються відповідні об’єкти або напряму у програмному контексті віртуальної машини застосунку користувача, або у реєстрі програмних залежностей, які отримає програмний застосунок користувача перед початком виконання його вихідного коду.

Таким чином, на противагу наявним фреймворкам автоматизованого тестування в програмній платформі Node.js, створений фреймворк реалізовує підміну залежностей компонентів системи користувача на етапі підготовки та налаштування системи, а не в процесі її роботи шляхом оновлення посилань, що є ненадійним методом ізолювання залежностей.

Подальші процеси роботи фреймворку є стандартизованими для будь-якого з наявних фреймворків автоматизованого тестування і охоплюють етапи виклику відповідного функціоналі програмного застосунку користувача, збір та аналіз статистик, перевірку результатів виконання сценаріїв тестування, а також оформлення результатів у вигляді деталізованого звіту про стан системи користувача та отриманих результатів тестування.

Останнім етапом роботи фреймворку тестування є очищення та вивільнення використаних ресурсів системи, завершення роботи програмного застосунку користувача та завершення роботи фреймворку тестування. Завершення роботи фреймворку тестування супроводжується передачею управління операційній системі користувача та сповіщення її сигналом статусу виконання сценаріїв тестування. Якщо хоча б один сценарій виявився хибним або сталась будь-яка незапланована помилка, то процес повинен завершитись з кодом помилки «1». В іншому випадку процес завершується з кодом «0», що означає що фреймворк успішно виконав усі заплановані задачі і досягнув поставленої мети тестування.

3.4 Висновки до розділу

У даному розділі представлено архітектурне рішення та реалізацію програмного забезпечення, яке відповідає поставленим у розділі 3.1 функціональним та нефункціональним вимогам.

Розроблене програмне забезпечення дозволяє виконувати автоматизоване модульне тестування програмного забезпечення за допомогою сценаріїв тестування, що розроблені на мові програмування, яка описана у 2 розділі цієї роботи.

Розроблений фреймворк тестування на відміну від наявних аналогів у програмній платформі Node.js, виконує сценарії тестування та вихідний код програмного застосунку клієнта у незалежних ізольованих середовищах тестування, що дозволяє не тільки контролювати процес тестування шляхом інверсії управління, а й надійно ізолювати компоненти системи для зменшення їх впливу один на одного.

Окрім того, розроблений фреймворк пропонує додаткові інструменти для проведення модульного тестування на кшталт модулю створення фіктивних даних та компонент, модулю доповідання результатів тестування тощо.

4 РЕЗУЛЬТАТИ ЕКСПЕРИМЕНТАЛЬНИХ ДОСЛІДЖЕНЬ

4.1 Мета та порядок досліджень

Для дослідження ефективності, лаконічності та виразності запропонованого методу опису сценаріїв автоматизованого тестування, а також швидкодії відповідного програмного застосунку проведення автоматизованого тестування, пропонується проведення низки експериментальних досліджень у порівнянні з наявними методами та засобами автоматизації тестування, а саме:

– порівняння лаконічності синтаксису сценарію модульного тестування, що написаний на розробленій формальній мові, з текстом опису сценарію тестування людською мовою за допомогою розрахунку відстані Левенштейна;

– розрахунок теоретичної та практичної виразності сценаріїв тестування, що описані на запропонованій формальній мові;

– порівняння сценарію модульного тестування, що написаний на розробленій формальній мові, з аналогічними сценаріями тестування, що описані за допомогою наявних фреймворків автоматизованого тестування програмної платформи Node.js, за кількістю ключових слів й символів для їх опису;

– порівняння швидкодії розробленого програмного застосунку виконання сценаріїв автоматизованого тестування з наявними засобами автоматизації тестування в програмній платформі Node.js.

Для проведення експериментальних досліджень застосовується апаратне забезпечення з наступними характеристиками:

– процесор: 2-ядерний Intel® Core™ i7-7500U з базовою частотою 2.90 ГГц;

– оперативна пам’ять: 16 ГБ SODIMM 2133 МГц;

– графічний процесор: NVIDIA GeForce GTX 960M 4 ГБ;

– операційна система: 64-розрядна Windows 10 Домашня.

4.2 Порівняння шляхом розрахунку відстані Левенштейна

Для порівняння лаконічності запропонованої формальної мови опису сценаріїв тестування з лаконічністю людської мови опису пропонується порівняти їх за допомогою розрахунку відстані Левенштейна – міри відмінності послідовності символів, що була запропонована радянським математиком Володимиром Левенштеном [27]. Вона визначається як мінімальна кількість односимвольних операцій вставки, видалення або заміни, що необхідні для перетворення однієї послідовності символів в іншу. Чим меншим є число необхідних операцій, тим більш схожими є дві послідовності символів. А отже, якщо сценарій автоматизованого тестування, що розроблений за допомогою запропонованої формальної мови, має малу відстань Левенштейна в порівнянні з аналогічним описом сценарію тестування людською мовою, то можна вважати, що запропонована формальна мова має наближену до людської мови лаконічність.

Для проведення експериментального дослідження було розроблено сценарій модульного тестування функції додавання чисел за допомогою запропонованої формальної мови (рисунок 4.1) та аналогічний сценарій модульного тестування (6), що описаний людською мовою за допомогою латинської абетки:

Unit test scenario "Sum two numbers" should call function "sum" from source file "./sum.js" with arguments 3 and 4 to test if correct answer 7 is returned as a result of the call. (6)

```
scenario `Sum two numbers` (
  sourceFile `./sum.js`,
  entity `sum`,
  cases (
    test `if correct answer 7 is returned` (
      steps (call `entity`, 3, 4)
            (expect `entity`, { returns: 7 })
    ),
  )
)
```

Рисунок 4.1 – Сценарій модульного тестування функції додавання на запропонованій формальній мові

Для розрахунку відстані Левенштейна застосовується метод матричного розрахунку з розміром матриці:

$$(n+1) \times (m+1),$$

де n та m – довжини порівнюваних рядків A та B відповідно. Для заповнення матриці застосовуються наступні формули:

$$m(a,b) = \begin{cases} 0, & a = b \\ 1, & a \neq b \end{cases},$$

$$\mathrm{D(i,j)} = \begin{cases} 0, & \text{якщо } i = 0, j = 0 \\ i, & \text{якщо } j = 0, i > 0 \\ j, & \text{якщо } i = 0, j > 0 \\ \min \begin{cases} D(i, j-1) + 1 \\ D(i-1, j) + 1 \\ D(i-1, j-1) + m(A_i, B_j) \end{cases}, & \text{якщо } j > 0, i > 0. \end{cases}$$

Таким чином, відстань Левенштейна між описом сценарію тестування на запропонованій формальній мові та людською мовою дорівнює 124 односимвольним операціям. Для порівняння на рисунку 4.2 наведено аналогічний сценарій модульного тестування, що розроблений із застосування фреймворку Jest, відстань Левенштейна якого у порівнянні зі сценарієм людською мовою дорівнює 128.

```
1 const sum = require('./sum.js');
2
3 describe('Sum two numbers', () ⇒ {
4   test('if correct answer 7 is returned', () ⇒ {
5     expect(sum(3, 4)).toBe(7);
6   });
7 });
```

Рисунок 4.2 – Сценарій модульного тестування функції додавання із застосуванням фреймворку Jest

Для проведення більш глибокого дослідження лаконічності синтаксису сценаріїв тестування пропонується розглянути складніший випадок сценарію модульного тестування: генерація масиву, що заповнений результатами виклику певної функції (рисунок 4.3).

```
function generateArray(length, generateFunction) {
  return Array.from({ length }, generateFunction);
}
```

Рисунок 4.3 – Вихідний код функції генерації масивів

Модульне тестування наведеної функції потребує передачі фіктивно-створеної функції в якості аргументи при тестуванні та перевірки, що результатом виконання цієї функції є масив певної довжини, що заповнений результатами виклику фіктивної функції. На рисунку 4.4 наведено сценарії модульного тестування функції, що описаний на запропонованій формальній мові, а на рисунку 4.5 наведено аналогічний сценарій тестування, що розроблений за допомогою фреймворку Jest. Опис аналогічного сценарію модульного тестування може бути представлений текстом (7), що описаний людською мовою за допомогою латинської абетки:

Unit test scenario "Generate prefilled array" should call function "generateFunction" from source file "./array.js" with 2 arguments: number 10 and a fake function that always returns number 1. Function "generateFunction" should return an array of size 10 that has each element equal number 1 as a result of the call. (7)

```
scenario `Generate prefilled array` (
  sourceFile `./array.js`,
  entity `generateFunction`,
  doubles (
    fakeFunction `fakeGenerateFn` ({
      implementation: () ⇒ 1
    })
  ),
  cases (
    test `function should return array` (
      steps (call `entity`, 10, fakeGenerateFn)
            (expect `entity`, {
              returns: {
                type: Array,
                length: 10,
                items: { equal: 1 }
              }
            })
    )
  )
)
```

Рисунок 4.4 – Сценарій модульного тестування функції "generateFunction" на запропонованій формальній мові

```
const generateFunction = require('./array.js');
const fakeGenerateFn = jest.fn(() => 1);

describe('Generate prefilled array', () => {
  test('function should return array', () => {
    const result = generateFunction(10, fakeGenerateFn);
    expect(result).toBeInstanceOf(Array);
    expect(result).toHaveLength(10);
    result.forEach(item => expect(item).toStrictEqual(1));
  });
});
```

Рисунок 4.5 – Сценарій модульного тестування функції "generateFunction" із застосуванням фреймворку Jest

Розрахунок відстані Левенштейна між описом сценарію тестування на запропонованій формальній мові та людською мовою показав, що необхідно 232 односимвольні операції для перетворення послідовності символів, тоді як для сценарію тестування, що описаний за допомогою фреймворку Jest, необхідно 280 операцій.

Таким чином, за результатами експериментальних досліджень можна вважати синтаксис запропонованої формальної мови більш лаконічним у порівнянні з аналогічним сценарієм тестування, що розроблений за допомогою фреймворку Jest. Оскільки синтаксис сценаріїв тестування, описаних у цій роботі фреймворків автоматизованого тестування, схожий між собою або аналогічний, то можна вважати, що запропонована предметно-орієнтована мова програмування краще підходить в якості мови опису сценаріїв автоматизованого тестування ніж наявні аналоги.

4.3 Порівняння шляхом підрахунку слів та символів

Іншим важливим фактором лаконічності та виразності запропонованої формальної мови є те, яка кількість слів та символів необхідна для опису сценарію тестування. Для прикладу розглянуто наведені вище сценарії тестування, а у таблиці 4.1 наведено підрахунок кількості необхідних слів та символів для опису відповідних сценаріїв модульного тестування. У таблиці 4.2 наведено кількість допоміжних символів (перенесення рядка та пробіли), що застосовуються для опису наведених сценаріїв

автоматизованого тестування, а у таблиці 4.3 наведено результуючий підрахунок необхідних слів та символів без врахування допоміжних символів.

Таблиця 4.1 – Підрахунок кількості необхідних слів та символів враховуючи символи перенесення рядка та пробіли

	Кількість необхідних слів			Кількість необхідних символів		
	Людська мова опису	Формальна мова опису	Опис за допомогою фреймворку Jest	Людська мова опису	Формальна мова опису	Опис за допомогою фреймворку Jest
Сценарій (6)	33	33	23	179	207	154
Сценарій (7)	50	52	35	317	457	375

Таблиця 4.2 – Кількість допоміжних символів для опису наведених сценаріїв автоматизованого тестування

	Людська мова опису	Формальна мова опису	Опис за допомогою фреймворку Jest
Сценарій (6)	32	57	25
Сценарій (7)	49	171	45

Таблиця 4.3 – Результати аналізу кількості необхідних слів та символів для опису сценаріїв тестування різними методами

	Кількість необхідних слів			Кількість необхідних символів		
	Людська мова опису	Формальна мова опису	Опис за допомогою фреймворку Jest	Людська мова опису	Формальна мова опису	Опис за допомогою фреймворку Jest
Сценарій (6)	33	33	23	179	150	129
Сценарій (7)	50	52	35	317	286	330

Оскільки тільки для людської мови пробіли та перенесення рядків відіграють певну семантичну та синтаксичну роль, то, відповідно, для запропонованої формальної мови, а також для мови опису сценаріїв тестування за допомогою фреймворку Jest, їх кількістю можливо знехтувати. Запропонована формальна мова опису сценаріїв тестування в порівнянні з аналогічними сценаріями, що описані за допомогою фреймворку Jest, хоч і потребує більшої кількості слів та символів у більшості випадків, але є найбільш наближеною до людської мови опису.

4.4 Порівняння швидкодії фреймворків тестування

Для визначення швидкодії розробленого програмного застосунку та його порівняння з наявними аналогами у програмній платформі Node.js, було застосовано програму GNU time версії 1.9 [28] з відкритим програмним кодом. Для проведення експерименту були використані сценарії модульного тестування з розділу 4.2 (рисунки 4.1, 4.2, 4.4 та 4.5), що були запущені у фреймворках в паралельному режимі, а результати експерименту представлені у вигляді графіку на рисунку 4.6.

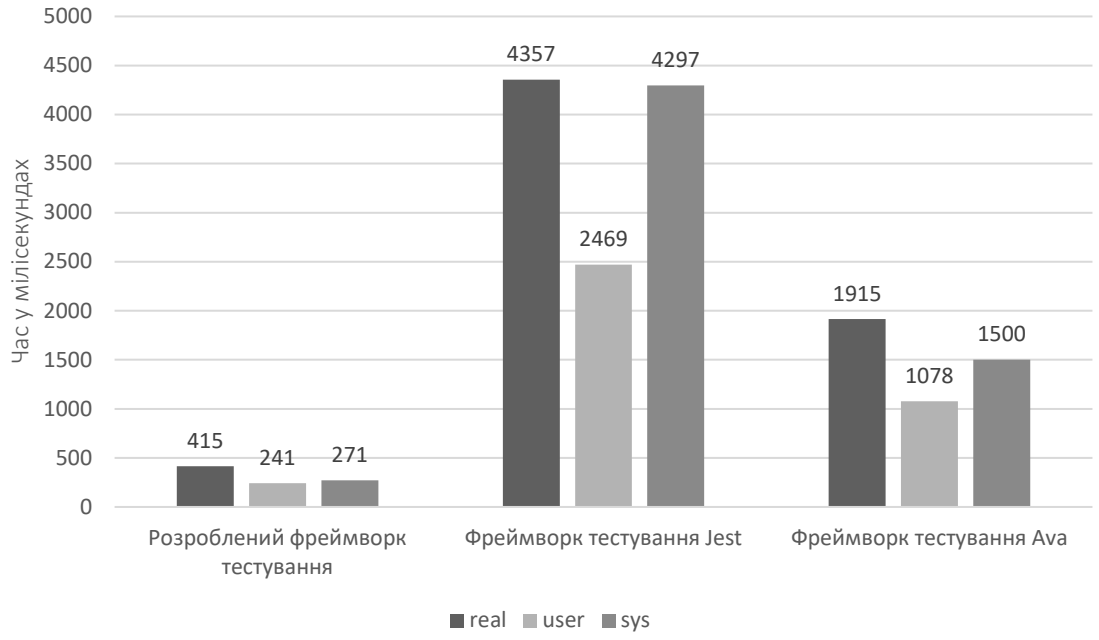


Рисунок 4.6 – Результати порівняння швидкодії програмних застосунків

Результати порівняння представлені трьома складовими:

– real time – часи виконання програми від запуску до припинення виконання;

– user time – час виконання програми на центральному процесорі;

– sys time – час очікування на виконання завдання операційною системою.

Як можна побачити з графіку на рисунку 4.6, швидкодія розробленого програмного застосунку значно більша, ніж у фреймворків автоматизованого тестування Jest та Ava. Таким чином, можна вважати, що розроблений програмний застосунок набагато краще підходить в якості фреймворку автоматизованого модульного тестування, аніж наявні аналоги.

4.5 Висновки до розділу

У даному розділі було проведено низку експериментальних досліджень характеристик запропонованого методу опису сценаріїв автоматизованого тестування та засобу для їх виконання.

В результаті проведених експериментальних досліджень можливо прийти до висновку, що запропонована предметно-орієнтована мова програмування наразі є найбільш наближеною до звичайної розмовної мови у порівнянні з наявними аналогами та найбільш вдало підходить для розв’язання задачі опису сценаріїв автоматизованого тестування.

В порівнянні з наявними аналогами розроблена формальна мова не потребує особливих навичок програмування для опису сценаріїв автоматизованого тестування, є простою та лаконічною, має виразний синтаксис, що легко сприймається при читанні та є інтуїтивно-зрозумілим при написанні.

Розроблений програмний застосунок для виконання сценаріїв автоматизованого тестування в порівнянні з аналогами у програмній платформі Node.js відрізняється значною швидкодією, що в середньому у 10 разів перевищує швидкодію аналогів.

ВИСНОВКИ

У даній магістерській дисертації розроблено методи та засоби для автоматизованого тестування програмного забезпечення, які підвищують ефективність процесу автоматизації тестування шляхом використання для опису сценаріїв тестування предметно-орієнтованої мови, що вирізняється лаконічністю та виразністю і є схожою на звичайну розмовну мову.

В процесі виконання роботи було досліджено наявні підходи до опису сценаріїв автоматизованого тестування. Виявлено, що більшість із них передбачає опис сценарію тестування безпосередньо мовою програмування, на якій написаний програмний застосунок, що тестується. Це вимагає від інженерів з тестування знання конкретної мови програмування і навичок програмування на ній. Використання ж деякими фреймворками предметно-орієнтованих мов для цих цілей також не розв'язує проблему, оскільки потребує додаткового опису таких сценаріїв програмним кодом.

Для розв'язання даної проблеми запропоновано удосконалений метод створення сценаріїв автоматизованого модульного тестування, який передбачає:

– структуризацію сценарію тестування у вигляді документу з ієрархічною структурою полів, що описують етапи тестування згідно зі стандартами проведення тестування;

– представлення окремих полів сценарію та тестових випадків у форматі модифікованих S-виразів, які є металінгвістичними абстракціями, заданими з використанням словникового запасу базової мови програмування;

– використання для формування такого сценарію спеціальної предметно-орієнтованої мови, яка максимально наближена до розмовної мови, однак має словниковий запас базової мови програмування.

В рамках реалізації даного методу розроблена предметно-орієнтована мова для формування структурованого сценарію тестування, поля і тестові випадки якого представляються металінгвістичними абстракціями, заданими з використанням

словникового запасу базової мови програмування. Дана формальна мова схожа на звичайну розмовну мову, проте залишається мовою програмування в контексті базової мови. Розроблена предметно-орієнтована мова є дочірньою до мови JavaScript, тому сценарії тестування, описані цією мовою, орієнтовані на обробку у програмному середовищі Node.js.

Для виконання сценарію тестування, написаного на розробленій предметно-орієнтованій мові, запропоновано метод трансляції, який, замість традиційних етапів, передбачає перетворення металінгвістичних абстракцій, в термінах яких задаються поля та тестові випадки сценарію тестування, в ієрархії взаємопов'язаних об'єктів та масивів з використанням підходів метапрограмування.

З метою реалізації запропонованих рішень був розроблений фреймворк для автоматизації модульного тестування для програмної платформи Node.js, який дозволяє виконувати сценарії тестування програмного забезпечення на мові JavaScript та вирізняється швидкодією проведення тестування у порівнянні з наявними аналогами.

Для забезпечення можливості проведення модульного тестування було запропоновано архітектурне рішення у вигляді підходу до ізоляції компонент програмної системи під час тестування. Сутність запропонованого підходу полягає у застосуванні віртуальної машини JavaScript-рушія Chrome V8 та спеціально налаштованого контексту виконання для забезпечення ізоляції компонентів системи. Такий підхід дозволяє проводити заміну компонентів системи до початку виконання програмного коду без внесення змін у вихідний код системи або оновлення глобального контексту виконання. В порівнянні з наявними підходами в фреймворках автоматизованого тестування програмної платформи Node.js, запропоноване рішення не впливає на роботу програмного застосунку та на результати тестування, оскільки кожний модуль програмного застосунку виконується в окремому незалежному середовищі віртуальної машини.

В ході роботи було також проведено дослідження характеристик запропонованих рішень. Результати експериментів показали, що ефективність розроблених методів та засобів для автоматизованого тестування в порівнянні з наявними методами та засобами у програмній платформі Node.js значно вирізняються у швидкодії, а запропонована предметно-орієнтована мова не має аналогів, проте дозволяє більш лаконічно описувати сценарії тестування.

Наукова новизна отриманих результатів полягає в удосконаленні методу створення сценаріїв автоматизованого тестування шляхом структуризації такого сценарію та представлення окремих полів сценарію й тестових випадків у форматі модифікованих S-виразів, які є металінгвістичними абстракціями, заданими з використанням словникового запасу базової мови програмування. Для формування такого сценарію була розроблена спеціальна предметно-орієнтована мова програмування, яка схожа на звичайну розмовну мову, а отже є простим та зрозумілим інструментом створення сценаріїв автоматизованого тестування.

Практична новизна полягає у створенні фреймворку модульного тестування для програмної платформи Node.js, що дозволяє проводити тестування на запропонованій предметно-орієнтованій мові та вирізняється швидкодією в порівняння з наявними аналогами.

Таким чином, усі поставлені завдання магістерської роботи виконані повною мірою, а поставлена мета – досягнута. Розроблені методи та засоби для автоматизованого тестування є зручними інструментами для реалізації модульного тестування у програмній платформі Node.js.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1) Pulse of the Profession 2019 [Електронний ресурс] // Project Management Institute, Inc. – Березень 2019. Режим доступу до ресурсу: https://www.pmi.org/learning/thought-leadership/pulse/pulse-of-the-profession-2019.

2) Brooks F. The Mythical Man-Month: Essays on Software Engineering / Frederick Brooks., 1995. – 336 с. – (Anniversary edition; 2).

3) Standard Glossary of Terms Used in Software Testing [Електронний ресурс] // International Software Testing Qualifications Board. – Грудень 2019. ред. ISTQB Glossary Working Group, Matthias Hamburg, Gary Mogyorodi. Версія 3.3. Режим доступу до ресурсу: https://glossary.istqb.org/.

4) Beizer B. Software Testing Techniques / Boris Beizer., 1990. – 580 с. – (2nd Edition).

5) When and what to automate in software testing? A multi-vocal literature review [Електронний ресурс]. – 2016. – Режим доступу до ресурсу: https://www.sciencedirect.com/science/article/abs/pii/S0950584916300702.

6) Mike Cohn. Succeeding with Agile. Software Development Using Scrum: книга // Mike Cohn. Addison-Wesley Professional. 2010. – 512 с.

7) Fields J. Working Effectively with Unit Tests // Jay Fields. Leanpub. 2015. – 347 с.

8) Приемы объектно-ориентированного проектирования. Паттерны проектирования // Э. Гамма, Р. Джонсон, Р. Хельм, Д. Влиссидес. Addison-Wesley. 2013. пер. с англ. А. Слинкин – 368 с. – (Библиотека программиста).

9) Vocke H. The Practical Test Pyramid [Електронний ресурс] / Ham Vocke // Martin Fowler Blog. – 2018. Режим доступу до ресурсу: https://martinfowler.com/articles/practical-test-pyramid.html.

10) Meszaros G. xUnit Test Patterns: Refactoring Test Code // Gerard Meszaros. Addison-Wesley. 2007. – 833 с.

11) Fowler M. Inversion of Control Containers and the Dependency Injection pattern [Електронний ресурс] / Martin Fowler. – 2004. – Режим доступу до ресурсу: https://www.martinfowler.com/articles/injection.html.

12) Fowler M. Integration Testing [Електронний ресурс] / Martin Fowler. – 2018. – Режим доступу до ресурсу: https://martinfowler.com/bliki/IntegrationTest.html.

13) Robinson I. Consumer-Driven Contracts: A Service Evolution Pattern [Електронний ресурс] / Ian Robinson // Martin Fowler Blog. – 2006. – Режим доступу до ресурсу: https://martinfowler.com/articles/consumerDrivenContracts.html.

14) OWASP Zed Attack Proxy Project [Електронний ресурс] – Режим доступу до ресурсу: https://wiki.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project.

15) National vulnerability database [Електронний ресурс] // The National Institute of Standards and Technology – Режим доступу до ресурсу: https://nvd.nist.gov/.

16) Node.js - Market Share & Web Usage Statistics [Електронний ресурс] // SimilarTech – Режим доступу до ресурсу: https://www.similartech.com/technologies/nodejs.

17) The State of JavaScript [Електронний ресурс]. – 2019. – Режим доступу до ресурсу: https://2019.stateofjs.com/testing/.

18) Abelson H. Structure and Interpretation of Computer Programs / H. Abelson, G. Sussman., 1996. – 883 с. – (Second edition).

19) McCarthy J. Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I [Електронний ресурс] / John McCarthy // MIT Press. – 1960. – Режим доступу до ресурсу: http://www-formal.stanford.edu/jmc/recursive/recursive.html.

20) ECMAScript 2021 Language Specification [Електронний ресурс]. – 2020. – Режим доступу до ресурсу: https://tc39.es/ecma262/.

21) Куликов С. Тестирование програмного обеспечения. Базовый курс. / Святослав Куликов. – Минск: Четыре четверти, 2017. – 312 с.

22) Chomsky N. Syntactic Structures / Noam Chomsky. – Berlin, New York: Mouton de Gruyter, 2002. – 118 с. – (Second edition).

23) ISO/IEC 14977:1996 (Information technology — Syntactic metalanguage — Extended BNF) [Електронний ресурс] – Режим доступу до ресурсу: https://www.iso.org/standard/26153.html.

24) Chomsky N. Aspects of the Theory of Syntax / Noam Chomsky., 1969. – 251 с.

25) The State of the Octoverse [Електронний ресурс] – Режим доступу до ресурсу: https://octoverse.github.com/.

26) Don't Block the Event Loop (or the Worker Pool) [Електронний ресурс] – Режим доступу до ресурсу: https://nodejs.org/en/docs/guides/dont-block-the-event-loop/.

27) Левенштейн В. Двоичные коды с исправлением выпадений, вставок и замещений символов / Володимир Йосипович Левенштейн. // Доклады Академий Наук СССР. – 1965. – С. 845–848.

28) GNU Time [Електронний ресурс]. – 2018. – Режим доступу до ресурсу: https://www.gnu.org/software/time/.

ДОДАТОК А

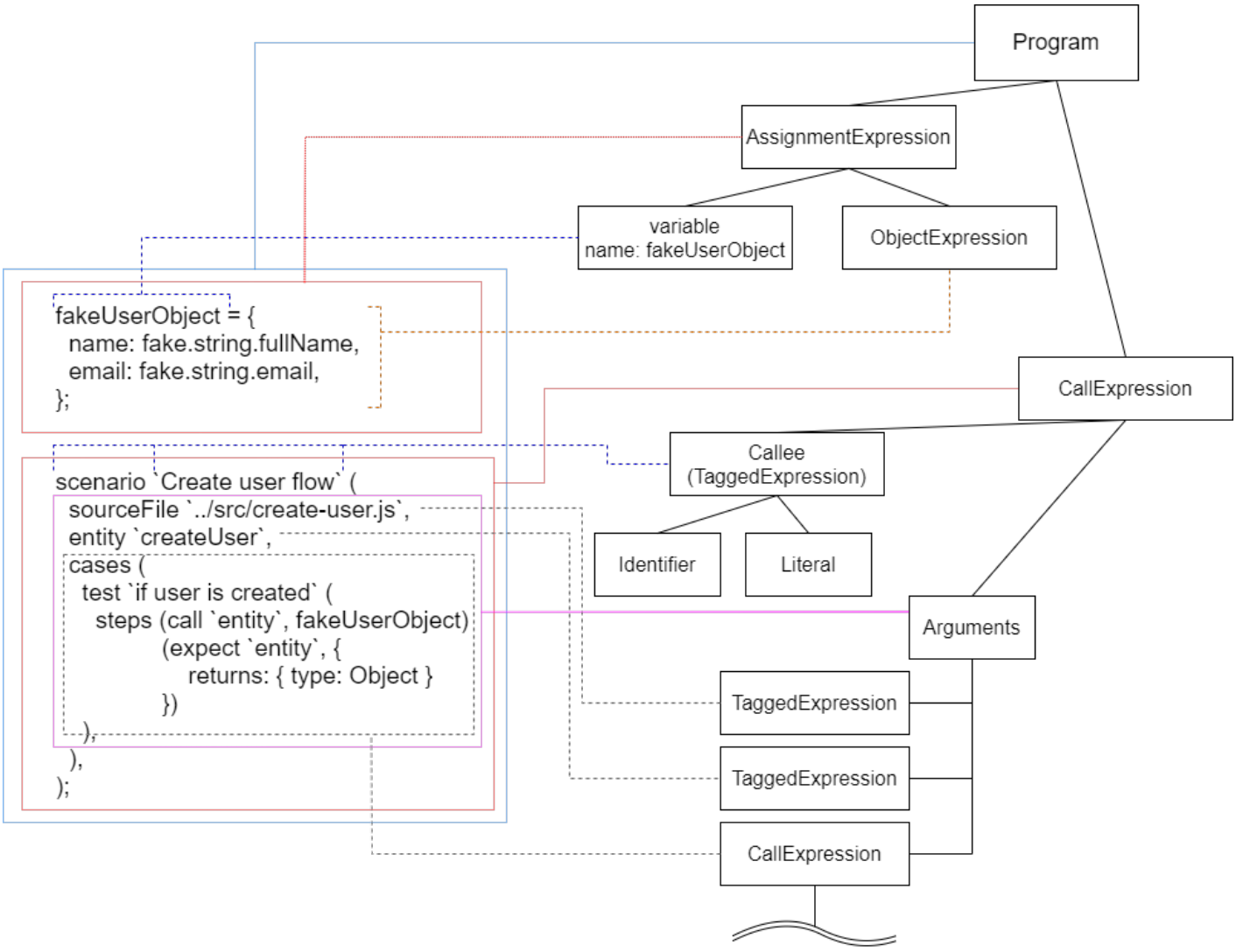
Формалізація створеної предметно-орієнтованої мови у розширеній нотації Бекуса-Наура

```
letter = "A" | "B" | "C" | "D" | "E" | "F" | "G"
      | "H" | "I" | "J" | "K" | "L" | "M" | "N"
      | "O" | "P" | "Q" | "R" | "S" | "T" | "U"
      | "V" | "W" | "X" | "Y" | "Z" | "a" | "b"
      | "c" | "d" | "e" | "f" | "g" | "h" | "i"
      | "j" | "k" | "l" | "m" | "n" | "o" | "p"
      | "q" | "r" | "s" | "t" | "u" | "v" | "w"
      | "x" | "y" | "z";
digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9";
digit_excluding_zero = "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9";
character = "[" | "]" | "{" | "}" | "(" | ")" | "<" | ">"
      | "'" | "`" | "=" | "|" | "." | "," | ";" | "/" | "\";
whitespace = " ";
variable_name = letter | "_", { letter | digit | "_" };
undefined = "undefined";
null = "null";
boolean = "true" | "false";
number = "" | "-", digit_excluding_zero, { digit }, ["."], { digit };
string = "" | '"', { letter | digit | character | whitespace }, "" | '"';
bigint = "" | "-", digit_excluding_zero, { digit }, "n";
symbol = "Symbol(", [string], ")";
array = "[", [{ containing, "," }] , "]";
object = "{", [{ string, ":", containing, "," }] , "}";
```

```
function = "(", [{ variable_name }] , ")", "=>", ("{" | "(", ""), [{ containing, (";", ",") }] , ("}" |
")", "");
containing = variable_name | undefined | null | boolean | number | string | bigint | symbol | array
| object;
variable = variable_name, "=", containing, ";";
(* Scenario Description *)
keyword = "description" | "entity" | "sourceFile" | "doubles" | "fakeFunction"
        | "before" | "after" |"call" | "saveReturnAs" | "cases" | "test"
        | "steps" | "expect" | "precondition" | "postcondition";
rule = keyword, "`", { letter | digit | character | whitespace }, "`", [{ rule_params }];
rule_params = "(", [{ rule | containing | "," }] ,")";
scenario = "scenario", "`", { letter | digit | character | whitespace } , "` (", [{ rule, "," }], ");";
grammar = [{ variable | scenario }];
```

ДОДАТОК Б

Абстрактне синтаксичне дерево сценарію автоматизованого тестування в загальному випадку



ДОДАТОК В

Схема-структурна послідовності роботи зі змінними у розробленому фреймворку автоматизованого тестування